UNIVERSITAT POLITÈCNICA DE CATALUNYA

FACULTAT D'INFORMÀTICA DE BARCELONA

PROJECTE FINAL DE CARRERA

ENGINYERIA EN INFORMÀTICA

Implementació en HDL d'un arbre binari de cerca auto-balancejat

Director:
Carlos ÁLVAREZ MARTÍNEZ
Departament del director:
Departament D'ARQUITECTURA DE COMPUTADORS

Autor:
Àlvar MERCADÉ IBÁÑEZ

Co-Director:
Daniel JIMÉNEZ GONZÁLEZ
Departament del co-director:
Departament D'ARQUITECTURA DE COMPUTADORS

FIB

UPC UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Implementació en HDL d'un arbre binari de cerca auto-balancejat

Àlvar Mercadé Ibáñez

27 de juny de 2017

Índex

Índex de figures

Llista d'algorismes

Índex de taules

Capítol 1

Introducció

1.1 Context del projecte

Amb la proliferació de les arquitectures multi-core i many-core, s'han emprat molts esforços en l'especificació i la implementació de nous models de programació, que facilitessin als desenvolupadors de programari la capacitat d'explotació de paral·lelisme en aquestes noves arquitectures; és a a dir, la capacitat que diversos processadors executin parts d'un programa simultàniament per tal de resoldre un determinat problema de còmput amb menys temps.

Un d'aquests models de programació en paral·lel és l'OmpSs[1]. Aquest model de programació es basa en la definició de tasques de còmput, fragments de programa que poden executar-se en paral·lel a d'altres, i les seves respectives dependències de dades en forma de dades d'entrada i dades de sortida o generades. D'aquesta manera, las diverses tasques es poden anar executant fora d'ordre, sempre i quan les seves dependències de dades estiguin resoltes.

Una problemàtica molt habitual en aquests models de programació cau en el fet que quan les tasques són molt fines, la planificació i gestió de tasques consumeix una quantitat de recursos i de temps d'execució que penalitza el rendiment final de l'aplicació. Com a solució a aquest problema, es va idear la creació de Picos, una implementació via maquinari del motor d'execució del model de programació OmpSs.

Tal i com s'explica en els articles [1, 2], es pot veure Picos com un co-processador que s'executa en una FPGA, que rep informació sobre la creació de noves tasques, amb les seves dependències i de la finalització d'aquestes. Amb aquesta informació va planificant i organitzant les tasques pendents d'execució i indica quina o quines tasques es troben ja preparades per a la seva execució. A la figura 1.1 a la pàgina següent es pot veure un exemple de la seva estructura hardware.

Un dels seus components és la memòria de dependències o DM. Aquesta memòria registra les adreces de dependències; és a dir, adreces de memòria que són generades per una tasca i que són consultades per altres tasques. Aquesta memòria està actualment implementada con una taula de dispersió o hash table 8-associativa, i presenta els següents problemes:

- No utilització de tota la capacitat de memòria disponible. Degut a la col·lisions provocades per a la funció de dispersió o hash.
- Temps d'espera excessius degut a stalls. Això està provocat pel problema anterior ja que quan una tasca

[1] https://pm.bsc.es/ompss

no es pot inserir en el sistema de la DM degut a una col·lisió, s'ha d'aturar Picos i s'ha d'esperar a que es buidi.

Per mitigar aquests problemes, es pretén la substitució de la DM actual per un arbre binari de cerca auto-balancejat. D'a- questa manera s'empraria sempre tota la capacitat de memòria i es reduirien el nombre de cicles perduts per les aturades del sistema.

Així doncs, l'objectiu d'aquest projecte que se us presenta consisteix en la implementació d'un arbre binari de cerca auto- balancejat en un llenguatge de descripció de maquinari per a re-emplaçar la memòria de dependències de Picos, un runtime per maquinari del model de programació OmpSs desenvolupat entre el Barcelona Supercomputing Center i el Departament d'Arquitectura de Computadors de la Universitat Politècnica de Barcelona.

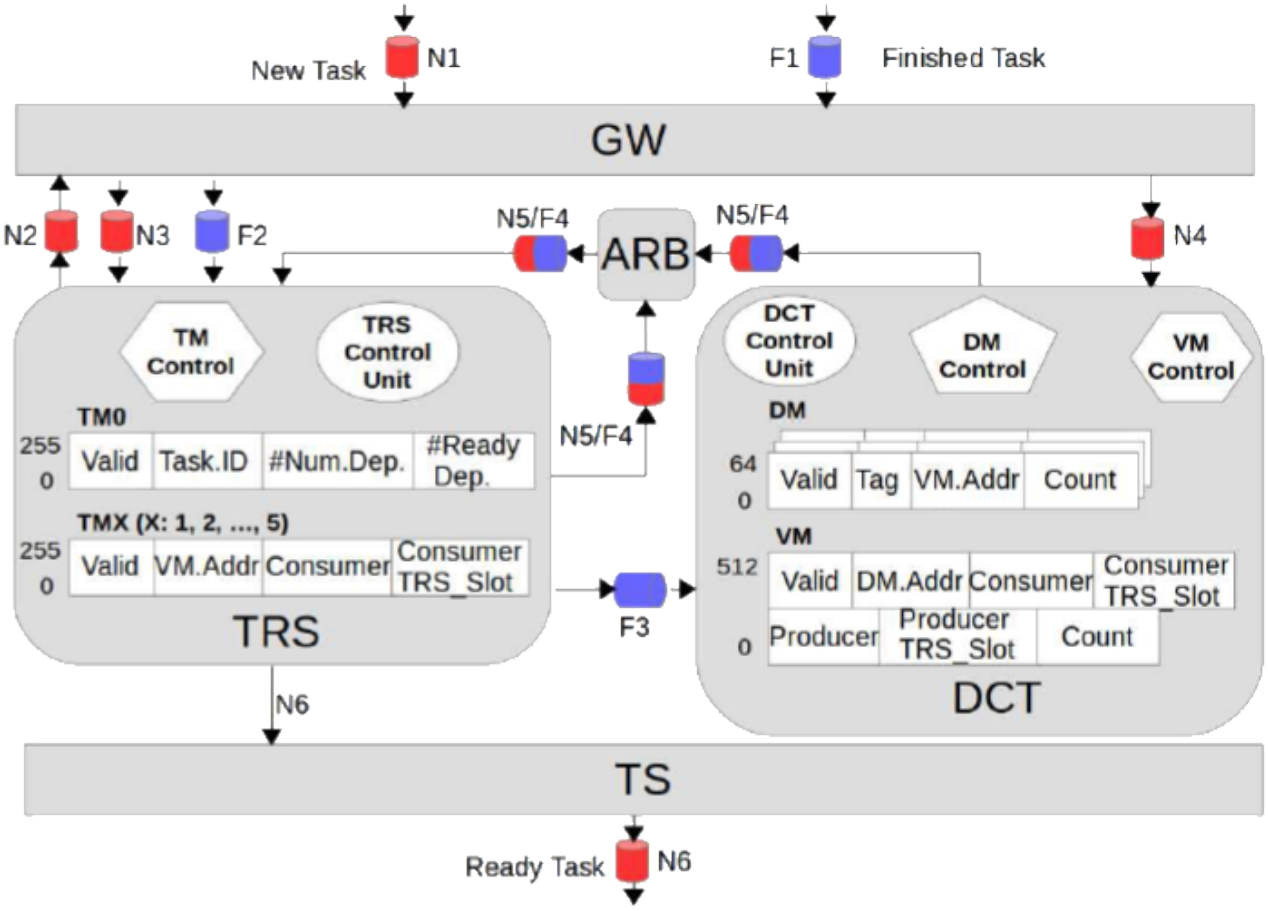

Figura 1.1: Picos

1.2 Treball relacionat

En aquest projecte es toquen diversos aspectes que es tracten en projectes d'investigació d'arreu del mon últimament tal i com es pot comprovar a la bibliografia on es pot trobar articles com [3, 4] que tracta sobre la implementacions d'algoritmes en llenguatges HDL per tal d'accelerar la seva execució sobre una FPGA, d'altres com [5, 6, 7, 8, 9] que tracten sobre estratègies diverses de cara a explotar el paral·lelisme de grau fi i d'altres com [4, 10, 11, 12] que tracten sobre l'arquitectura de les FPGA. Per tant, queda demostrat la utilitat i l'atractiu que aquest projecte i les seves possibilitats de cara a futur.

Capítol 2

FPGAs & llenguatges de descripció de maquinari

2.1 FPGA: Una aproximació

Un dispositiu *Field-programmable gate array* (d'ara en davant FPGA, les seves sigles en anglès) és un dispositiu electrònic que té la capacitat de reconfigurar-se al llarg de la seva vida a petició de l'usuari. Aquesta capacitat permet al dispositiu portar a terme una mateixa tasca que un computador tradicional però amb un major rendiment i/o menor consum energètic. A la figura 2.1 a la pàgina 13 es pot veure com és una placa amb una FPGA.

Les configuracions s'especifiquen; o es programen segons es prefereixi dir, emprant textos escrits en un HDL[1]. A l'actualitat els dos llenguatges majoritaris d'aquesta família con el llenguatge *Verilog* i el llenguatge *VHDL*. Entre els camps de la ciència i l'enginyeria on l'ús d'aquests dispositius és més notori són els següents:

- Defensa i Enginyeria Aeroespacial
  - Comunicacions
  - Armament
  - Aviònica
- Multimèdia (Imatge/Vídeo i So)
  - Processat imatges en alta o molt alta resolució.
  - Processat de senyal digital d'àudio.
- Comunicacions
  - Transmissions via senyals òptiques.
  - Transmissions via ones de ràdio.
- Seguretat

[1] HDL: HARDWARE DESCRIPTION LANGUAGE, Llenguatge de descripció de maquinari.

  - Processat d'imatges.
  - Criptografia.
- Medicina
  - Ultra-sons.
  - Tomografies computeritzades.
  - Rajos X.
  - Imatges de ressonàncies magnètiques.
- Enginyeria Industrial
  - Control de motors.
- Automoció
  - Processat d'imatges en temps real.
  - Comunicació entre vehicles.
- HPC (Computació d'Alt Rendiment) & Centres de processament de dades
  - Xarxes (enrutadors i commutadors).
  - Balancejadors de càrrega.
  - Servidors.

El funcionament d'una FPGA radica en dos components primaris, la LUT (de *Lookup table*) és una petita memòria programable que rep unes senyals d'entrada i et genera la sortida. El fet que aquesta memòria sigui programable és la base sobre la qual es pot implementar la lògica d'un disseny determinat. En aquesta arquitectura una LUT es pot comportar com qualsevol porta lògica que fos necessari. Les Luts es poden connectar amb biestables per formar el que s'anomenen Slice o cel·les lògiques. Aquestes s'agrupen formant blocs de lògica complexa o CLB, i s'interconnecten mitjançant la matriu d'enrutament. Aquesta matriu, que és el segon component bàsic d'una FPGA, és la que permet connectar diverses cel·les lògiques per crear circuits lògics més complexes.

2.2 Descripció de maquinari: El llenguatge VHDL

Per a la implementació d'aquest projecte s'havia de triar entre dos llenguatges de programació específics per a la programació de dispositius FPGA: Verilog i VHDL.

El primer dels dos llenguatges, el Verilog - combinació dels mots *verification* i *logic* -, és un llenguatge estandarditzat sota la norma IEEE 1364[2][13] de l' *Institute of Electrical and Electronics Engineers*. Creat per Prabhu Goel, Phil Moorby and Chi-Lai Huang and Douglas Warmke entre els anys 1983 i 1984, va ser dels primers llenguatges creat per a la especificació i disseny de maquinari i va ser el primer àmpliament adoptat arreu del món.

És un llenguatge que és més fàcil de cara l'entrada de gent nova a la programació de hardware gràcies al fet que té moltes característiques copiades del llenguatge de programació de software C tals com els mots per

[2] http://ieeexplore.ieee.org/document/1620780/

definir les estructures de control de flux i la prioritat dels operadors. Com a punt negatiu d'aquest llenguatge té el seu fluix sistema de tipus de dades, com passa amb el llenguatge C, que tot i fer que sigui molt fàcil obtenir un disseny sintetitzat -el que en el món de la programació del programari en diríem una compilació- provoca que sigui molt fàcil la introducció d'errors en el disseny que no es detectin fins a la seva simulació o execució en una FPGA. Per tant pot resultar frustrant per a la gent que s'introdueixi en aquest món.

Per l'altra banda, el llenguatge VHDL, estandarditzat sota la norma IEEE 1076[3][14], va ser un llenguatge creat a petició del Departament de Defensa dels Estat Units d'Amèrica a l'any 1981 sota les següents premisses:

1. El llenguatge havia de ésser capaç d'especificar la lògica dels sistemes sense ambigüitats i amb la capacitat d'adaptar els dissenys a les noves tecnologies que anessin apareixent amb el pas del temps sense haver de modificar el procès de disseny, eliminant d'aquesta manera l'obsolescència tecnològica.

2. El llenguatge resultant havia de ser comprensible pels éssers humans per tal de facilitar el seu manteniment i la seva modificació.

El llenguatge resultant és molt més textual en comparació amb el llenguatge VHDL i té un sistema de comprovació de tipus més rígid. La conseqüència d'aquests fets és, per a la gent principiant pot resultar més dur la seva programació en la mesura que s'ha d'introduir més quantitat de text per a especificar la mateixa unitat lògica, però, contràriament al llenguatge Verilog, un cop sintetitzat un disseny, aquest no contindrà errors degut a ambigüitats o a errors de tipatge de les dades.

Per tant, havent fet un petit estudi sobre les característiques dels dos principals llenguatges ja esmentats i recordant experiències de programació viscudes a la Facultat d'Informàtica de Barcelona, es decideix que el llenguatge de disseny triat per a la realització d'aquest projecte és el VHDL per ser un llenguatge més segur i robust per a compensar el meu poc recorregut en el camp de la programació de hardware.

Un disseny en el llenguatge VHDL consisteix de dos parts. La primera és la definició de la entitat, que es defineix un mòdul amb un nom i la seva relació de senyals d'entrada i/o de sortida. La segona part és l'arquitectura. En aquesta part es defineix com es comporta aquest mòdul, és a dir, s'explica la seva lògica interna: com llegeix les senyals d'entrada, i com genera senyals de sortida a partir de les senyals d'entrada i del seu estat intern i com es modifica aquest últim. En una arquitectura ens podem trobar amb construccions presents en llenguatges de programació de software com assignacions, operacions aritmeticològiques, blocs condicionals, dades estructurades, etc.

[3] http://ieeexplore.ieee.org/document/4772740/

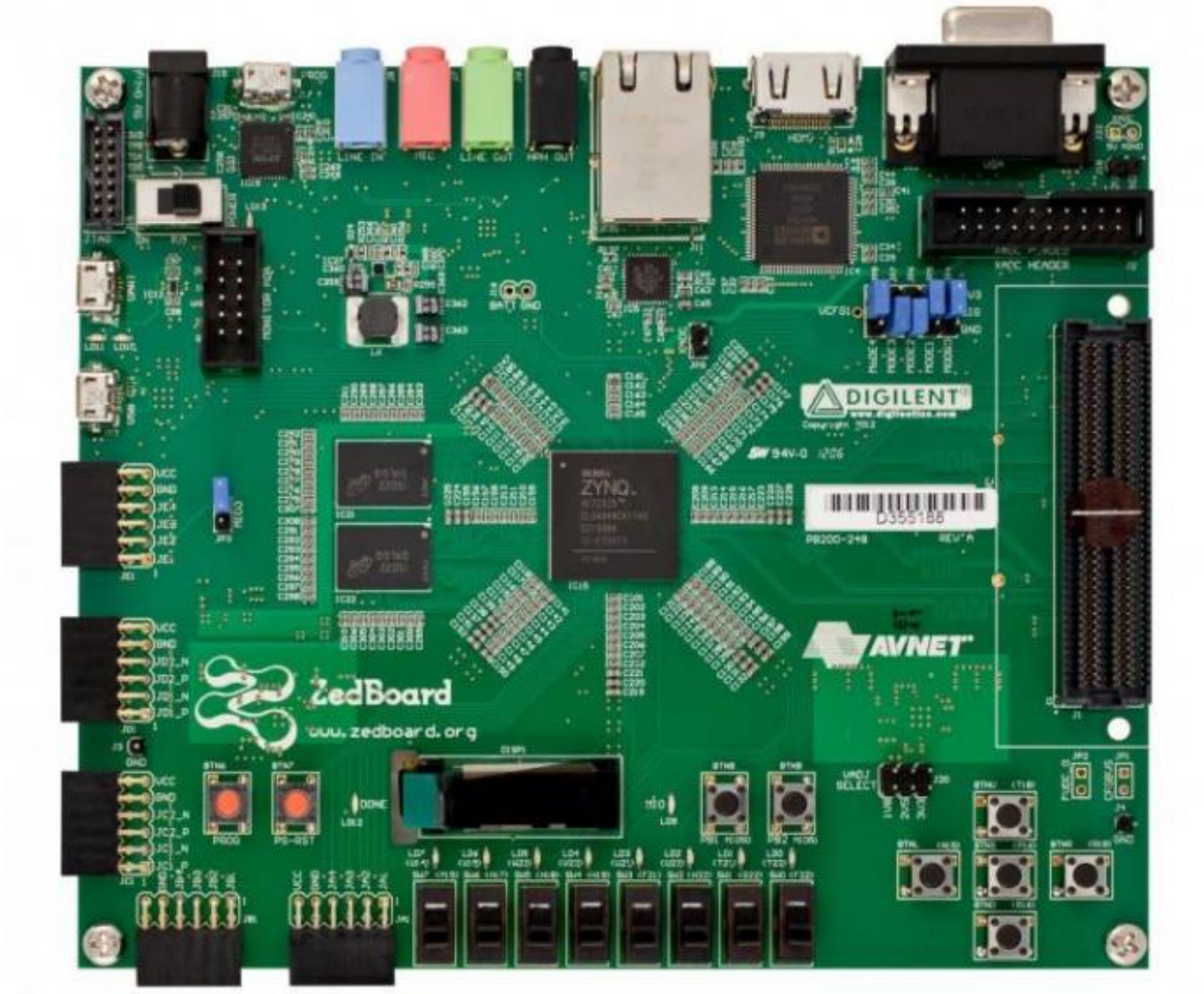

Figura 2.1: Placa Zedboard amb FPGA Zynq de la casa Xilinx

Capítol 3

Arbres de cerca auto-balancejats: L'arbre Red-Black

3.1 Definició

Un arbre binari de cerca auto-balancejat, és una estructura de dades mutable de tipus conjunt on si poder realitzar 3 operacions bàsiques:

1. Cerca: donat un arbre i un valor, t'afirma o nega l'existència d'aquest valor dins l'arbre.
2. Inserció: donat un arbre i un valor, modifica l'arbre de manera que aquest valor passa a formar part si i només si, aquest valor no formava prèviament part de l'arbre.
3. Eliminació: donat un arbre i un valor, s'extreu el valor de l'arbre si i només si, aquest en formava part de l'arbre.

Per tal de dur a terme aquestes operacions de manera eficient, la informació a emmagatzemar s'organitza en diverses entitats anomenades nodes que es troben connectat mitjançant arestes i que es relacionen de la següent manera:

- Des d'un node qualsevol es pot accedir directament a 2 nodes com màxim.
- Aquests dos nodes fills els anomenarem fill esquerre i fill dret.
- Existeix un únic node que pot accedir, a través d'aquest, a qualsevol altre node de l'arbre, però que no s'hi pot arribar des de cap altre node de l'arbre. A aquest node l'anomenem l'arrel de l'arbre.
- Per cada node que no sigui l'arrel de l'arbre, només existeix un únic camí amb origen l'arrel i amb destinació el node en qüestió.
- Donat un node qualsevol, tots els elements que s'hi pot accedir a través del seu fill esquerre són d'un valor menor al valor del node i, per simetria, tots els element que s'hi pot accedir a través del seu fill dret són nodes amb valors superiors a a quest node.

- Per tal de satisfer aquest punt, cal que la informació sigui d'una tipologia on s'hi pugui establir una notació d'ordre, ja sigui de manera natural (que la tipologia de la informació ja inclogui un mecanisme d'ordenació natural com és el cas de les informacions de tipus numèrica o alfabètica, o de manera arbitrària explicitant el mecanisme d'ordenació.
- Cada cop que es modifica aquest arbre, es re-estructura de manera que la seva alçada, és a dir la distància màxima entre el node arrel i qualsevol node de l'arbre entenent aquesta distància com el nombre de nodes que formen part del camí que porta des de l'arrel fins al node, es manté mínima. Aquesta cota és de l'ordre del logaritme en base 2 del nombre de nodes que formen part de l'arbre.

L'avantatge d'aquesta estructura de dades és que permet dur a terme les 3 operacions bàsiques en temps logarítmic, seguint terminologia matemàtica. Això implica, per exemple, que si tenim un arbre que conté un miler de milions (1.000.000.000) d'elements, només caldrà visitar uns trenta (30) elements per saber si un determinat element existeix o no.

3.2 Estructures alternatives

L'arbre Red-Black, no és l'únic arbre binari de cerca balancejat. Existeixen altres estructures de dades que compleixen el mateix objectiu que l'arbre Red-Black, però amb filosofies o estratègies diferents. A continuació s'indiquen una sèrie d'aquestes estructures i els motius pels quals no han estat l'estructura escollida per desenvolupar aquest projecte.

L'arbre Splay és un arbre que té com a particularitat que els elements recentment visitats o consultats s'estructuren més a prop de l'arrel, propiciant d'aquesta manera que properes consultes és facin amb menys temps. Aquesta estructura es descarta, perquè en aquest projecte no té sentit una operació de consulta sense cap efecte, ja que sempre una operació en el nostre esquema comportarà potencialment una modificació en l'arbre. Un altre motiu per descartar aquesta estructura és que no assegura un temps logarítmic $O(log_2 n)$ per a cada operació.

Una altra estructura és l'arbre scapegoat, que té com a particularitat que no afegeix cap dada addicional sobre els nodes respecte a l'arbre binari de cerca convencional. Però el cost és que les operacions per re-equilibrar l'arbre tenen cost lineal, $O(n)$, i, per tant, massa costoses en còmput pel que es desitja.

L'arbre 2-3 o altres implementacions del B-arbre és una estructura alternativa on en el nodes intermedis, poden contenir múltiples valors. Aquesta estructura es descarta per la seva dificultat per ser el primer projecte sobre FPGA del projectista.

L'arbre AVL és l'arbre que té les operacions de consulta més eficients respecte a altres arbres. El motiu són les re-estructuracions estrictes que tenen lloc després de modificar l'arbre per tal d'assegurar que la seva alçada és mínima. Això comporta que les operacions d'inserció i esborrat siguin molt costoses i que no sigui adequat aquesta estructura per situacions on l'arbre estigui en constant modificació. És per aquest últim motiu, que es descarta aquesta estructura.

Les principals avantatges de l'arbre Red-Black, són el poc sobre-cost en termes de memòria respecte a l'arbre binari de cerca (només afegeix un bit pel color i un apuntador addicional sobre el node pare)

3.3 Estructura

L'estructura de dades anomenada arbre Red-Black, és una de les vàries estructures de dades que implementen l'arbre binari de cerca auto-balancejat. Aquesta variant té, com a característica que la defineix, la particularitat

que a cada node de l'arbre s'afegeix un bit d'informació addicional anomenat color. Aquest color únicament por contenir dos valors diferents: Vermell (Red) o Negre (Black); d'aquí el nom de la variant[1]. I com pot un sol bit d'informació assegurar que l'alçada de l'arbre, el de la figura 3.1 per posar-ne un exemple, es mantingui en $O(log_2 n)$? Doncs seguin un seguit de regles que es detallen a continuació un cop es realitzi cap modificació sobre l'arbre:

1. Tot no o bé és de color negre o bé és de color vermell.
2. El node arrel sempre és de color negre.
3. Cada una de les fulles (nodes sense fills esquerre ni dret) és de color negre.
4. Si un node és vermell, llavors tots dos fills són de color negre.
5. Tots els camins des del node arrel fins a les fulles conté el mateix nombre de nodes negres.

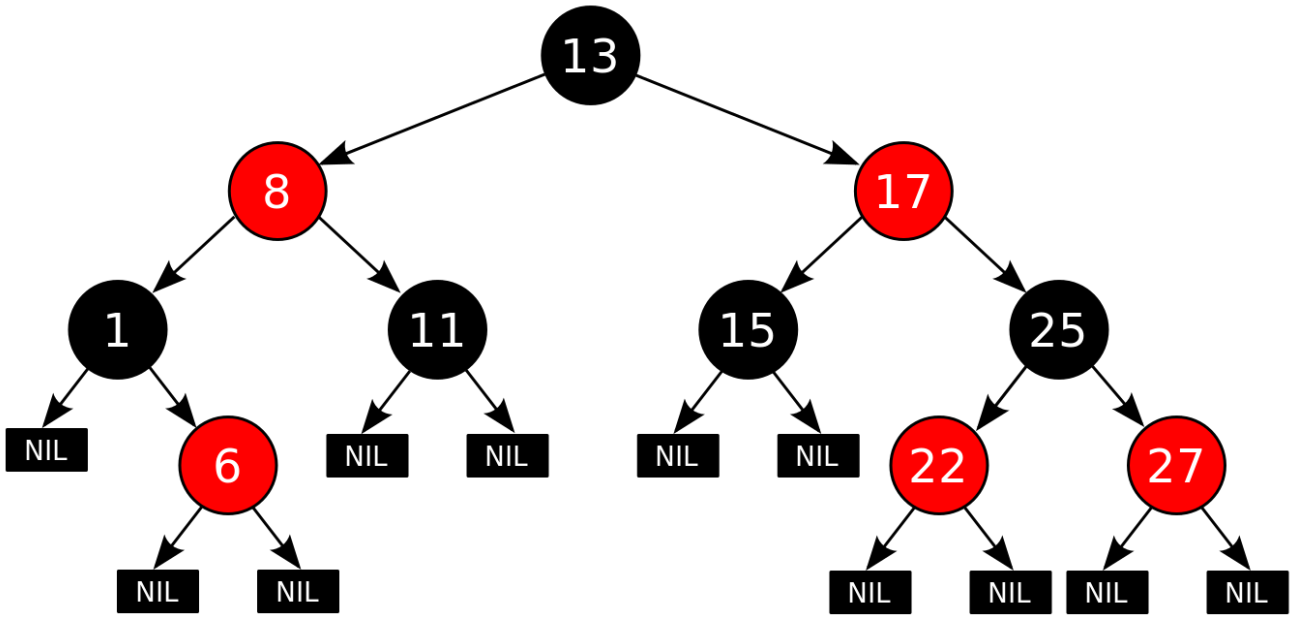

Figura 3.1: Exemple d'arbre Red-Black

L'arbre Red-Black, com tots els altres arbres de cerca balancejats o no, respon a dues operacions bàsiques:

1. Cerca/Inserció: Donat un element e, el cercarà dins l'estructura de l'arbre i, en cas que no el trobi, l'inserirà i dit nombre passarà a formar part de l'estructura de l'arbre, en cas que l'element en qüestió ja formés part de l'arbre, l'operació finalitza sense portar a terme cap modificació sobre l'arbre.
2. Eliminació: Donat un element e, el cercarà dins de l'estructura de l'arbre i, en cas que l'element en qüestió ja formés part de l'estructura de l'arbre, l'extreu i es re-estructura mantenint vigent les seves propietats.

[1] S'utilitzen aquests colors perquè eren els que millor es mostraven en la versió impresa de l'article escrit l'any 1978 A Dichromatic Framework for Balanced Trees per Guibas i Sedgewick utilitzant una de les impressores làser a color de que disposaven a Xerox Park.

Per tant, podem es pot definir aquesta estructura de dades tal i com es pot veure en l'algorisme 3.1 emprant el llenguatge C++:

Algorisme 3.1 Definició de l'estructura de dades arbre *Red-Black*

```
class RedBlack
{
    private:
        enum Color {roig, negre};
        struct RedBlackNode
        {
            RedBlackNode *pare,*esquerra,*dreta;
            int valor;
            Color c;
        }
        RedBlackNode *arrel = nullptr;
        // Aquestes 4 funcions s'expliquen més endavant
        void rotar_esquerra(RedBlackNode*);
        void rotar_dreta(RedBlackNode*);
        void comprovar_insertar(RedBlackNode*);
        void comprovar_esborrar(RedBlackNode*);
    public:
        bool cerca(const int);
        void insertar(const int);
        void esborrar(const int);
}
```

3.4 Operacions

3.4.1 Cerca

L'algoritme de la cerca d'un element dintre d'un arbre *Red-Black* és idèntic al corresponent en un arbre binari de cerca convencional.

Primer de tot, ens situem a l'arrel de l'arbre i llavors, mentre ens trobem amb un node mirem si el valor emmagatzemat en el node és el que estem buscant. En cas contrari ens dirigirem cap al fill esquerra o el dret en funció si el valor que estem buscant és menor al valor present en el node.

En el llenguatge C++, es pot escriure l'algoritme de cerca tal i com es veu en l'algorisme 3.2 a la pàgina següent:

Com que en cada iteració descartem explorar la meitat del sub-arbre que es visita, es constata que el cost d'aquesta operació és $O(log_2\, n)$, sent n el nombre de valors presents a l'arbre.

Algorisme 3.2 Implementació en C++ de l'operació de cerca

```
bool RedBlack::cerca(const int valor)
{
    RedBlackNode *n = arrel;
    while (n != nullptr)
    {
        if(n->valor == valor) {
            // Valor trobat
            return true;
        } else if(valor < n->valor) {
            n = n->esquerra;
        } else {
            n = n->dreta;
        }
    }
    // Si hem arrivat al final vol dir que no l'hem trobat
    return false;
}
```

3.4.2 Inserció

L'operació d'inserció, tal i com es pot veure en l'algorisme 3.3 a la pàgina següent, es conforma de dues parts. En la primera, tenim l'algoritme de la inserció en un arbre de cerca bàsic lleugerament modificat. Els canvis vénen de gestió del color del nou node; negre en el cas que el nou node s'instal·li en l'arrel de l'arbre o roig en cas contrari. En la segona part, només s'executarà en cas que el nou node no sigui l'arrel de l'arbre i que el seu node predecessor sigui roig.

Aquest cas comporta la violació d'una de les normes exposades en el capítol anterior (la norma 4 per ser més exacte). Per evitar aquesta situació es procedirà a la re-estructuració mitjançant rotacions i re-pintat de nodes. Les rotacions, tal i com es veuen en la figura 3.2 a la pàgina 20 i en l'algorisme 3.4 a la pàgina 21, són operacions que alteren l'ordre entre nodes pròxims, però mantenint les propietats d'un arbre binari de cerca convencional. Existeixen dues operacions de rotació en funció del sentit del gir de la mateixa.

Les comprovacions i les corresponents actuacions sobre l'arbre es duran a terme en el procediment *comprovar_insertar*, que es pot veure en l'algorisme 3.5 a la pàgina 26. En aquest procediment, s'analitzen una sèrie de casos possibles que es poden donar i es duen a terme els canvis corresponents. Un concepte que s'utilitzarà en aquests casos és el concepte del *germà*. Aquest germà és l'altre node fill del pare d'un node determinat. En els següents casos el germà farà referència al germà del node pare del nou node.

3.4.2.1 Cas 1: El cas dels germans rojos

Aquest cas es dóna quan el node pare i el seu germà són nodes rojos. Per tant, i per mantenir l'equilibri de nodes en forma de nombre de nodes negres que hi ha des de l'arrel fins als nodes fulla, el que es porta a terme és un canvi de colors dels nodes pare, germà i el seu propi pare, tal i com es veu en els dos casos de la figura 3.3 a la pàgina 22.

Un cop s'han re-pintat els nodes s'ha eliminat el conflicte entre el nou node inserit (25) i el seu node pare

Algorisme 3.3 Codi de la inserció d'un element en C++

```
void RedBlack::insertar(const int valor)
{
    RedBlackNode *n = arrel, *p = arrel, *nou;
    while (n != nullptr) {
        if(n->valor == valor) {
            // Si l'hem trobat, no l'hem d'insertar
            return;
        } else if(valor < n->valor) {
            p = n;
            n = n->esquerra;
        } else {
            p = n;
            n = n->dreta;
        }
    }
    // Si hem arrivat fins al final, vol dir que no l'hem trobat i que, per tant,
        s'ha d'insertar el valor dins l'arbre.
    nou = new RedBlackNode;
    nou->valor = valor;
    nou->esquerra = nullptr;
    nou->dreta = nullptr;
    // p apunta al node pare del nou node o nullptr en cas que sigui  el nou node
        l'arrel de l'arbre
    if(p == nullptr) {
        // L'arrel sempre és un node negre
        arrel = nou;
        arrel->c = negre;
    } else {
        node->pare = p;
        node->c = roig;
        if(nou->valor < p->valor) p->esquerra = nou;
        else p->dreta = nou;
        if(p->c == roig) comprovar_insertar(nou);
    }
}
```

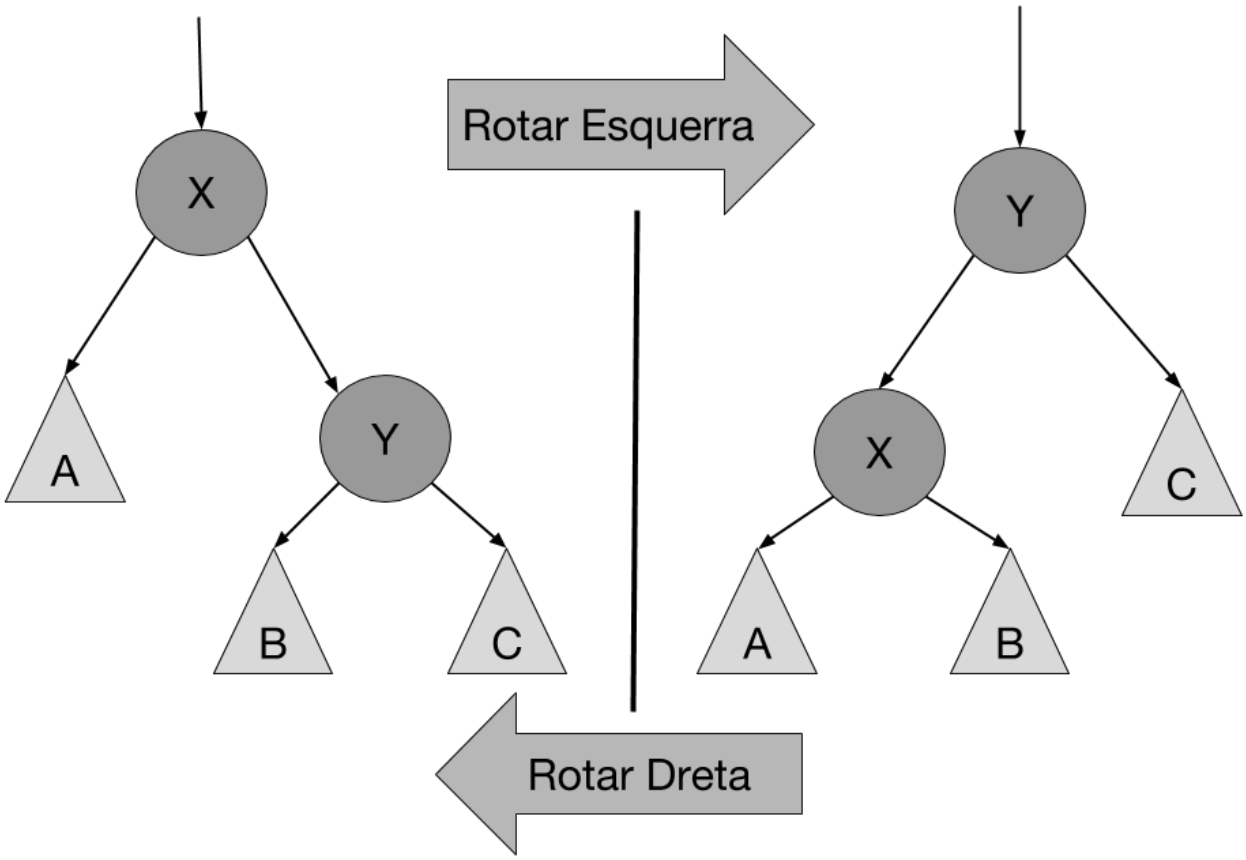

Figura 3.2: Esquema gràfic de com funcionen les rotacions en un arbre

(30), però al canviar de color el node superior (20), s'ha pogut violar la norma referent al fet que el node arrel sempre és de color negre o podria ser que aquest node pengés d'un altre node i que fos de color roig. Per tant el que s'ha de fer es realitzar les comprovacions des d'aquest node.

3.4.2.2 Cas 2: El cas dels 3 nodes en línia

Aquest cas es dóna quan tant el nou node com el node pare són o bé fill esquerra o bé fill dret. D'aquesta manera es troben alineats amb el node *anterior*[2] formant una línia recta.

Per resoldre aquest conflicte de dos nodes rojos consecutius el que es porta terme és una rotació sobre el node anterior fent que el node pare sigui el nou node anterior, a continuació aquests dos nodes s'intercanvien els seus colors per trencar el conflicte de dos nodes rojos i es manté l'equilibre de nodes negres, tal i com es veu en la figura 3.4 a la pàgina 24. Per tant un cop fetes aquestes accions, es pot donar les comprovacions per finalitzades.

[2]Em refereixo al node pare del node pare del nou node.

Algorisme 3.4 Codi de les rotacions en C++

```
void RedBlack::rotar_esquerra(RedBlackNode *n) {
    RedBlackNode *m = n->dreta;
    m->pare = n->pare;
    n->pare = m;
    n->dreta = m->esquerra;
    n->dreta->pare = n;
    m->esquerra = n;
    if(arrel = n) arrel = m;
    else if(m->padre->esquerra = n) m->padre->esquerra = m;
    else m->pare->dreta = m;
}

void RedBlack::rotar_dreta(RedBLackNode *n) {
    RedBlackNode *m = n->esquerra;
    m->pare = n->pare;
    n->pare = m;
    n->esquerra = m->dreta;
    n->esquerra->pare = n;
    m->dreta = n;
    if(arrel = n) arrel = m;
    else if(m->padre->esquerra = n) m->padre->esquerra = m;
    else m->pare->dreta = m;
}
```

3.4.2.3 Cas 3: El cas de la ziga-zaga

Aquest cas es dóna quan el node anterior, el pare i el nou node formen un ziga-zaga. Dit d'una altra manera, quan el pare és el fill esquerra i el nou node fill dret o viceversa. En aquest cas, per resoldre la situació el que s'ha de dur a terme és una doble rotació tal i com es descriu gràficament en la figura 3.5 a la pàgina 25. Primerament sobre el node pare, fent que el nou node pare sigui el nou node i després sobre el node anterior, promocionant el nou node com a arrel d'aquest sub-arbre. Finalment, s'intercanvien els colors entre el fins ara node anterior i el nou node. Com que el nou node superior té el mateix color que l'anterior arrel del sub-arbre no s'introdueix cap altre violació dels invariant dels arbres *Red-Black*, es pot donar les comprovacions per finalitzades.

3.4.3 Esborrat

La operació d'esborrar, com es pot veure en l'algorisme 3.6 a la pàgina 27, d'un element dintre d'un arbre *Red-Black* consta de dues parts: la primera, i tal i com passava amb la operació d'inserció, és una versió lleugerament modificada de la mateixa operació corresponent a l'arbre binari de cerca convencional. Aquesta modificació es basa en el fet de memoritzar el color i el node que s'esborrarà o re-emplaçarà el node a esborrar, i que mantindrà el color del node a esborrar. La segona part és una comprovació, ja que en el fet que s'hagues esborrat o re-emplaçat un node de color negre estarem violant un dels invariants dels arbres *Red-Black*, en

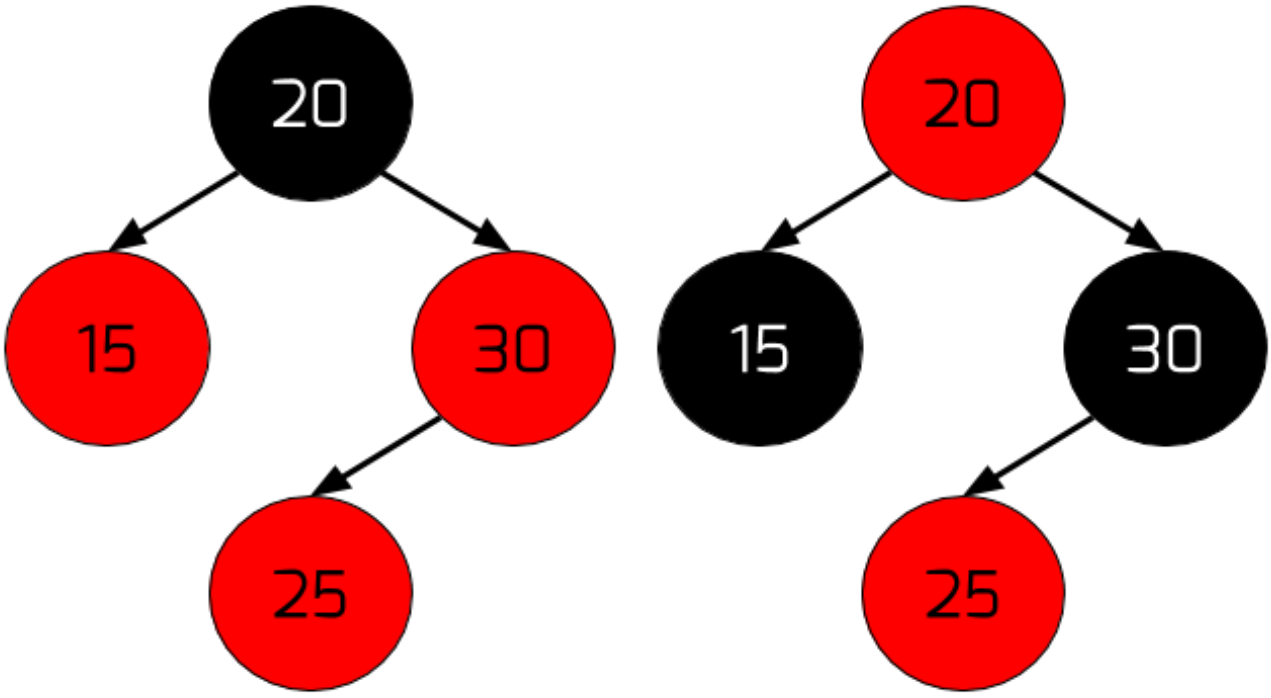

Figura 3.3: Cas 1 de les comprovacions d'una inserció.

concret a l'invariant 5. En aquesta comprovació, que es porta a terme en la subrutina *comprovar_esborrar* escrita en l'algorisme 3.7 a la pàgina 32, es tracta de re-estructurar el sub-arbre corresponent per tal de poder pintar un node roig com a negre per tal de mantenir l'invariant com a vàlid seguint una sèrie de casos.

Una forma de veure com es procedirà a la resolució d'aquest conflicte es basa en el fet d'imaginar-se que el node *m* de l'algorisme anterior té el que es pot dir un *doble-negre.* Com si es tractés que quantitat d'un líquid, pots tenir el node buit (roig), ple (negre) o amb excés de líquid (doble-negre) i que per tant, aquest excés de líquid s'ha de traspassar a un altre node que tingui el dipòsit buit (pintar un node que inicialment era de color roig a color negre), mantenint en tot moment l'equilibri amb la resta de l'arbre.

3.4.3.1 Cas 1: El cas del germà roig

Aquest cas es dona quan el germà del node amb *doble-negre* és de color roig. En aquest cas es procedeix a la rotació sobre el pare del node de tal manera que el que fins ara era el seu germà passi a ser *l'avi* del node amb *doble-negre* tal i com es pot veure en la figura 3.6 a la pàgina 28.

Un cop feta aquesta transformació, mantenim el *doble-negre* en la seva posició i es passa a comprovar quins dels següents 3 casos es pot passar a aplicar a continuació.

3.4.3.2 Cas 2: El cas dels nebots negres

Aquest cas es dona quan els dos fills del germà del node amb *doble-negre* són de color negre. en aquest cas, veient que cap rotació ni doble rotació pot propiciar un escenari per desfer el *doble-negre;* es procedirà a elevar un negre del node amb *doble-negre* i un *negre* del seu germà i reiterarem amb les comprovacions des del node pare, com es pot veure en la figura 3.7 a la pàgina 29.

Si el node pare és de color roig, llavors es pot re-pintar de color negre i desapareix el conflicte per *doble-negre*. En cas contrari es comença a realitzar les comprovacions per casos de nou sobre aquest node fins a resoldre el conflicte.

3.4.3.3 Cas 3: El cas del nebot pròxim roig

Aquest cas es dóna quan el node és fill esquerra i el fill esquerra del seu germà és roig o tots dos són fills drets. Com es pot veure en la figura 3.8 a la pàgina 30, es pot aprofitar aquesta situació per a realitzar una doble rotació, primer sobre el germà de tal manera que el nebot roig sigui el nou germà i una segona sobre el node pare del node amb *doble-negre,* resultant en que el que inicialment era el nebot roig ara passa a ser el node pare del pare del node amb *doble-negre.* Acte seguit aquest node es queda amb el color que té el node pare i el node pare passa a ser de color negre a resultes de propagar-li el negre sobrant del node amb *doble-negre.*

Com que ha desaparegut el node amb *doble-negre,* es pot donar les comprovacions per finalitzades.

3.4.3.4 Cas 4: El cas del nebot llunyà roig

Aquest cas es dóna quan el node amb *doble-negre* és fill esquerra i el fill dret del germà és un node roig o viceversa. En aquest cas podem produir el mateix escenari final que en el cas anterior, però solament amb una sola rotació sobre el node pare del node amb *doble-negre,* com es pot veure en la figura 3.9 a la pàgina 31. Fent que el germà sigui el node pare del node pare es pot traspassar els colors entre els nodes que inicialment eren els nodes germà, pare i nebot *llunyà* de manera que es pot resoldre el *doble-negre.*

Tanmateix passava amb el cas anterior, com que es resol el conflicte del *doble-negre*, es poden donar les comprovacions per finalitzades.

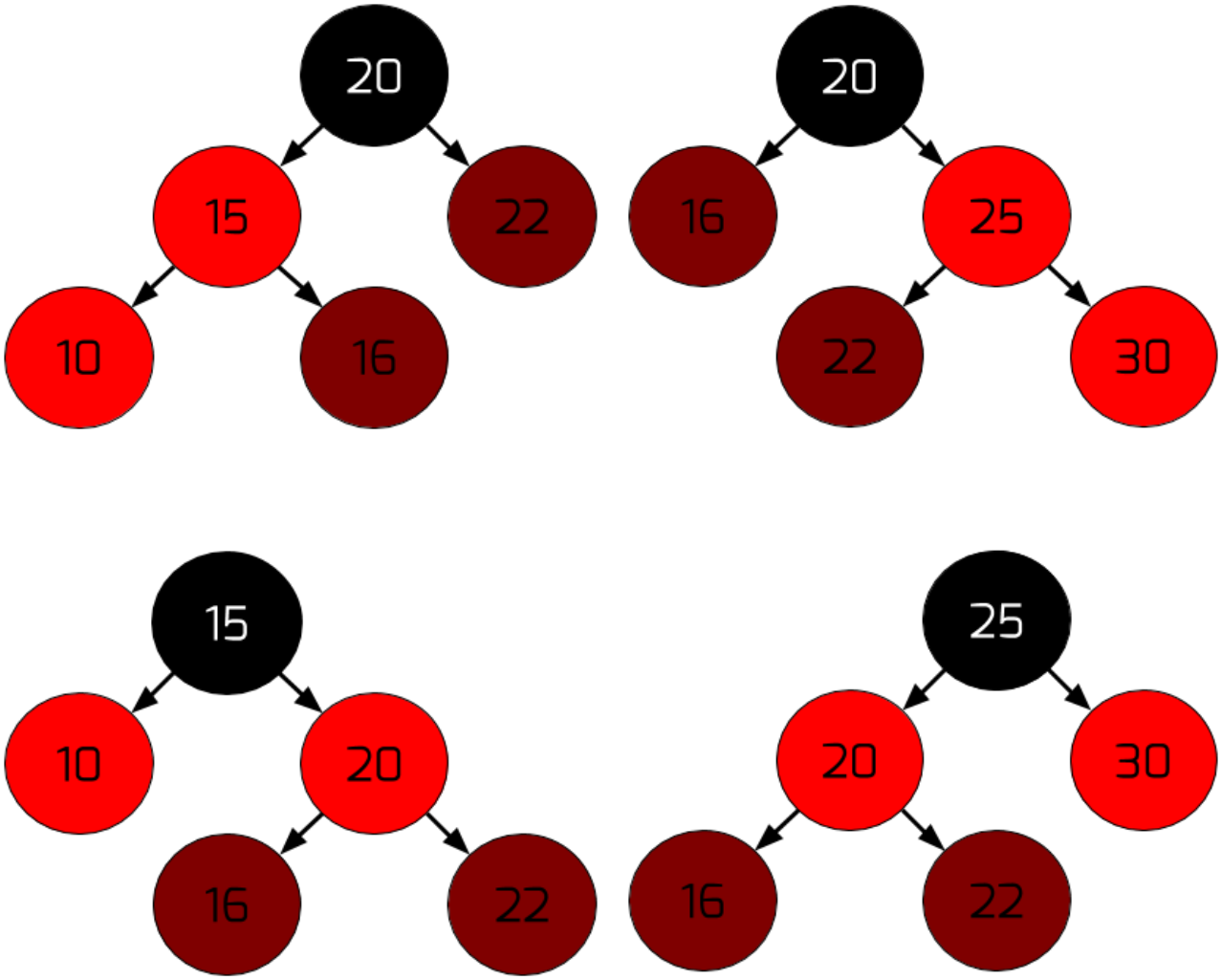

Figura 3.4: Cas 2 de les comprovacions d'una inserció.

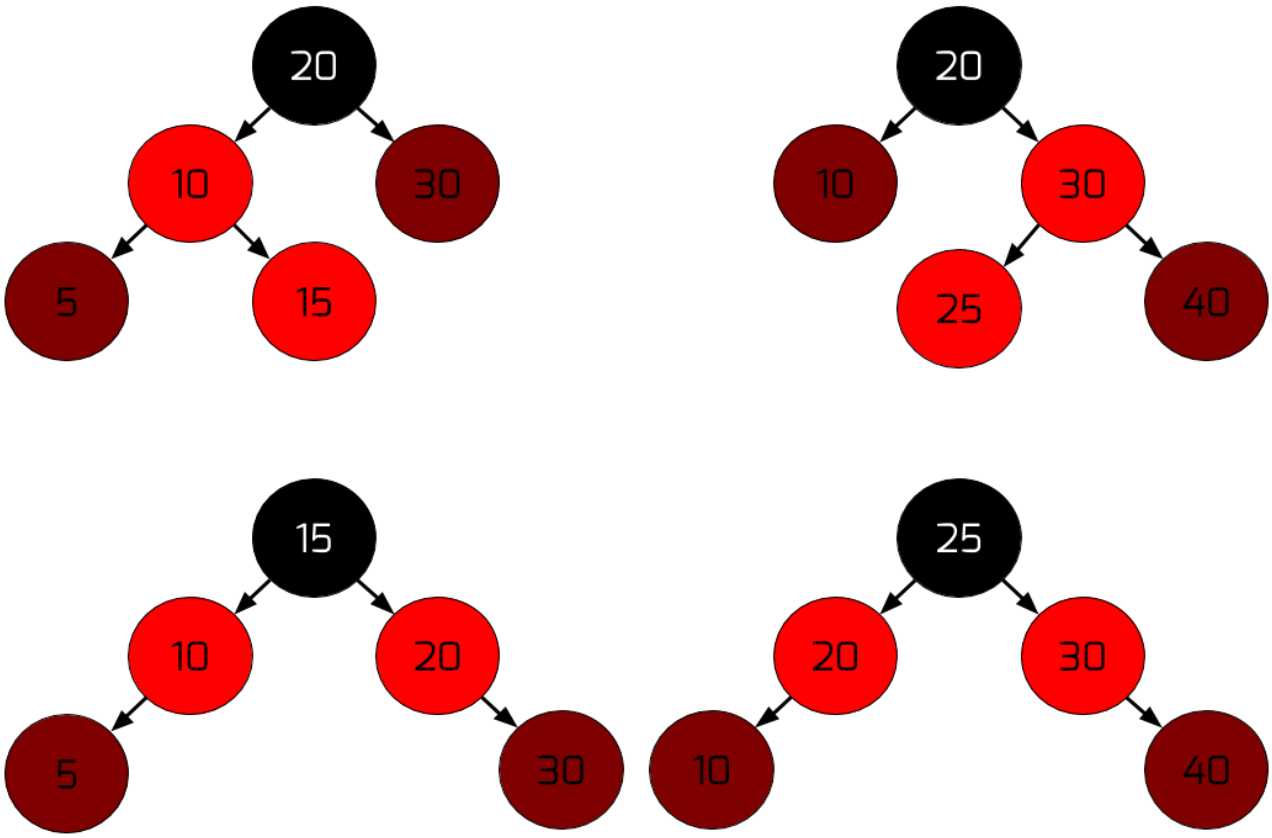

Figura 3.5: Cas 3 de les comprovacions d'una inserció

Algorisme 3.5 Procediment per a la comprovació post-inserció en C++

```
void RedBlack::comprovar_insertar(RedBlackNode* n) {
    RedBlackNode *germa;
    if(n->c = negre)
        arrel->c = negre;
    else {
        if(n->pare = n->pare->pare->esquerra) {
            germa = n->pare->pare->dreta;
            if(germa->c = roig) { // Cas 1
                n->pare->c = negre;
                germa->c = negre;
                n->pare->pare->roig;
                comprovar_insertar(n->pare->pare);
            } else {
                if(n = n->pare->esquerra) { // Cas 2
                    n->pare->c = negre;
                    n->pare->pare->c = roig;
                    rotar_dreta(n->pare->pare);
                } else { // Cas 3
                    n->pare->pare->c = roig;
                    n->c = negre;
                    rotar_esquerra(n->pare);
                    rotar_dreta(n->pare);
                }
            }
        } else {
            germa = n->pare->pare->esquerra;
            if(germa->c = roig) { // Cas 1
                n->pare->c = negre;
                germa->c = negre;
                n->pare->pare->roig;
                comprovar_insertar(n->pare->pare);
            } else {
                if(n = n->pare->dreta) { //Cas 2
                    n->pare->c = negre;
                    n->pare->pare->c = roig;
                    rotar_esquerra(n->pare->pare);
                } else { // Cas 3
                    n->pare->pare->c = roig;
                    n->c = negre;
                    rotar_dreta(n->pare);
                    rotar_esquerra(n->pare);
                }
            }
        }
    }
}
```

Algorisme 3.6 Esborrat d'un element a l'arbre

```
void RedBlack::esborrar(const int valor) {
    RedBlackNode *n = arrel, *m;
    Color color_m;
    while(n != nullptr) {
        if(n->valor == valor) {
            // Element trobat
            color_m = n->c;
            if(n->esquerra = nullptr and n->dreta == nullptr) {
                m = n->pare;
                if(m->esquerra == n) m->esquerra = nullptr;
                else m->dreta = nullptr;
            } else if(n->esquerra = nullptr) {
                m = n->dreta;
                m->pare = n->pare;
                if(m->pare->esquerra == n) m->pare->esquerra = m;
                else m->pare->dreta = m;
            } else if(n_dreta == nullptr) {
                m = n->esquerra;
                m->pare = n->pare;
                if(m->pare->esquerra == n) m->pare->esquerra = m;
                else m->pare->dreta = m;
            } else {
                m = n->dreta;
                while(m->esquerra != nullptr) m = m->esquerra;
                color_m = m->c;
                m->pare->esquerra = m->dreta;
                m->dreta->pare = m->pare;
                m->pare = n->pare;
                m->esquerra = n->esquerra;
                m->dreta = n->dreta;
                m->c = n->c;
            }
            delete n;
            if(color_m == negre) comprovar_esborrar(m);
        } else if(valor < n->valor)
            n = n->esquerra;
        else
            n = n->dreta;
    }
}
```

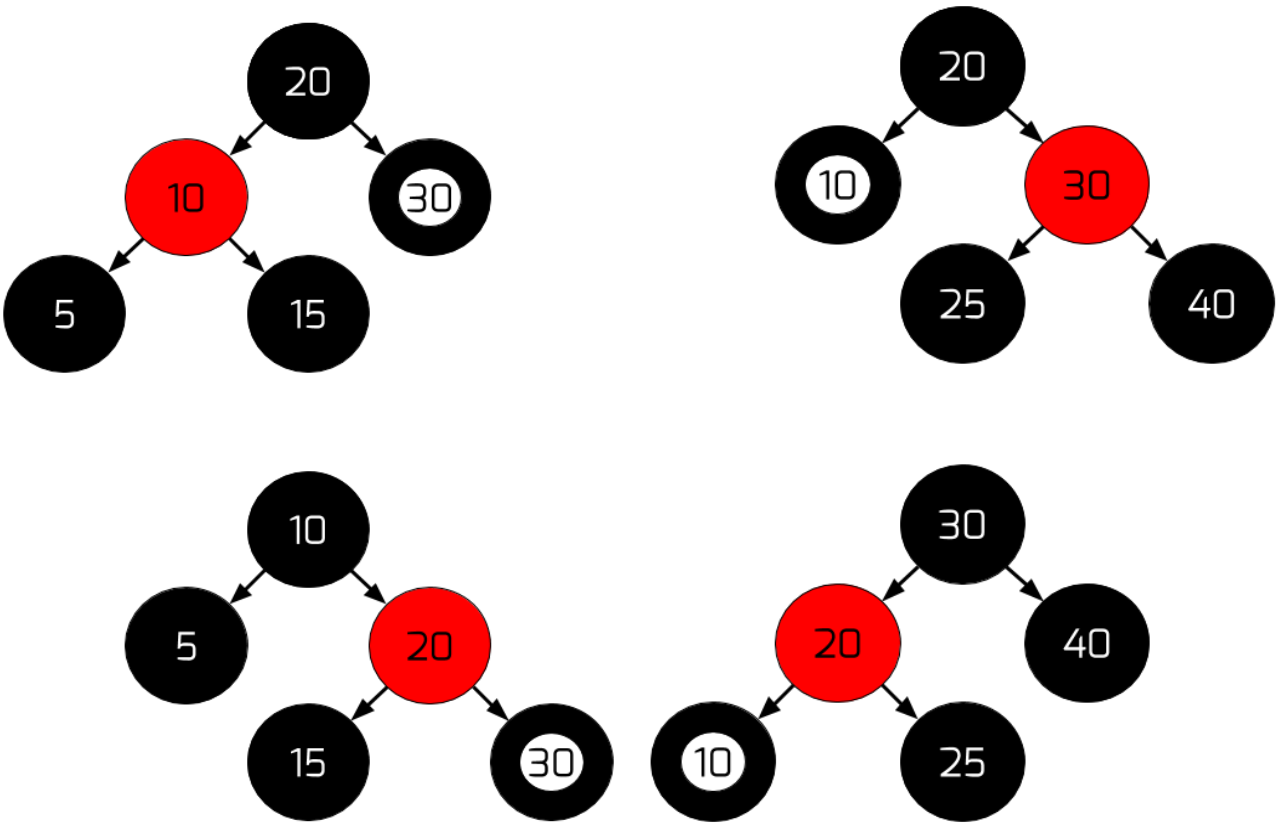

Figura 3.6: Cas 1 de la comprovació post-esborrat

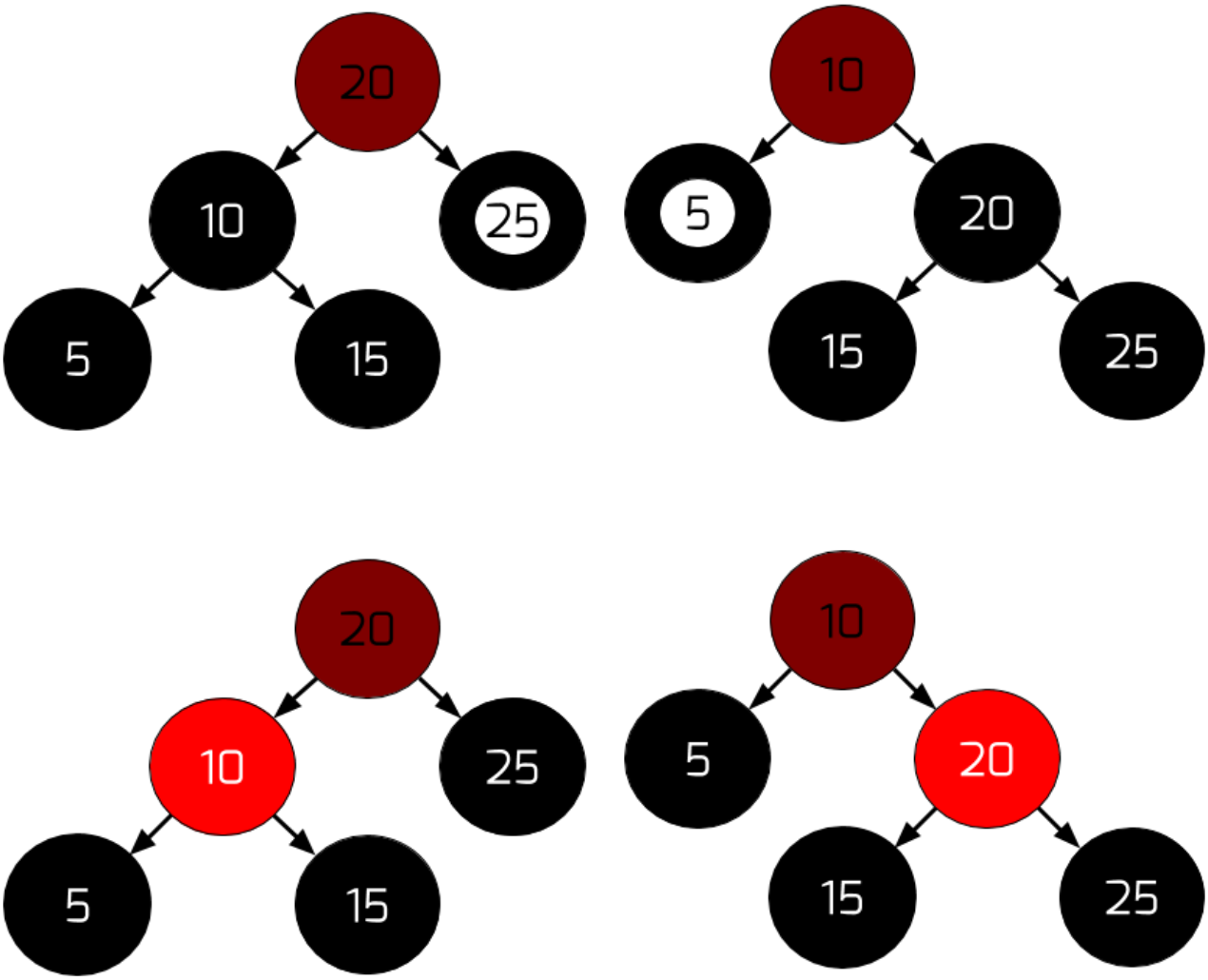

Figura 3.7: Cas 2 de la comprovació post-esborrat

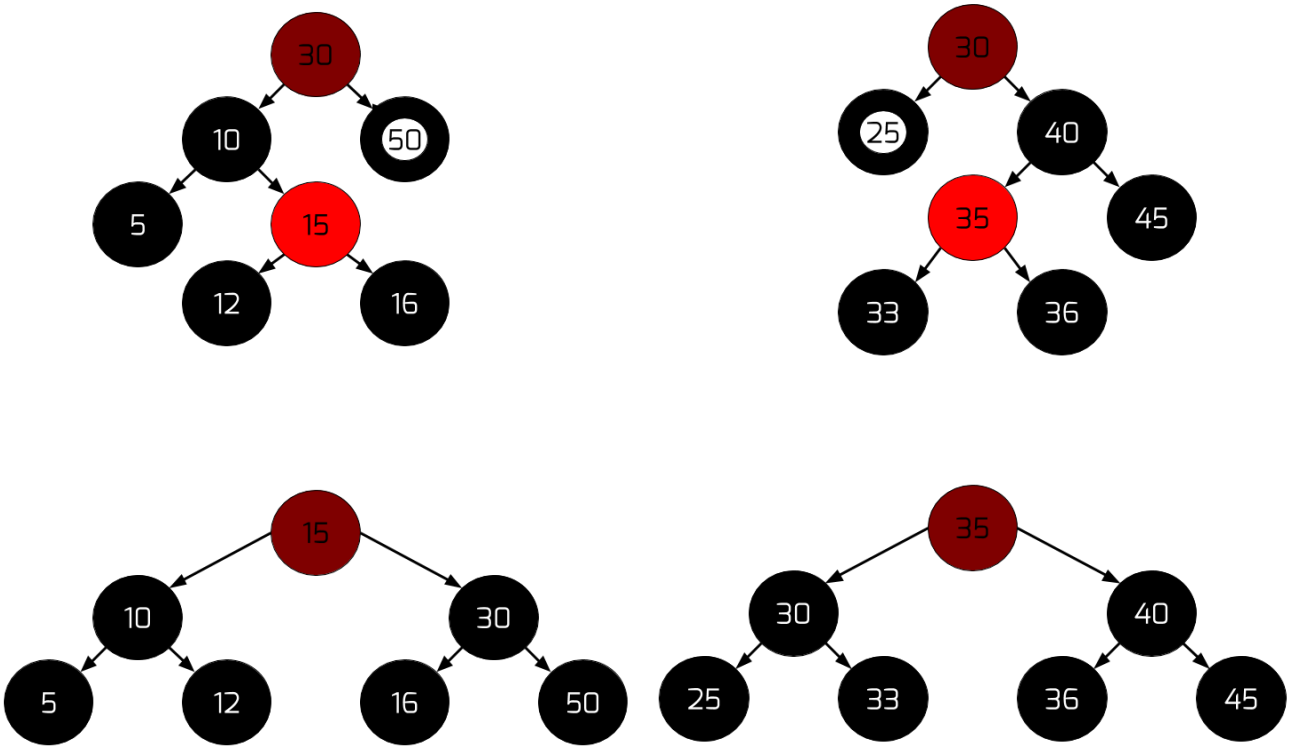

Figura 3.8: Cas 3 de la comprovació post-esborrat

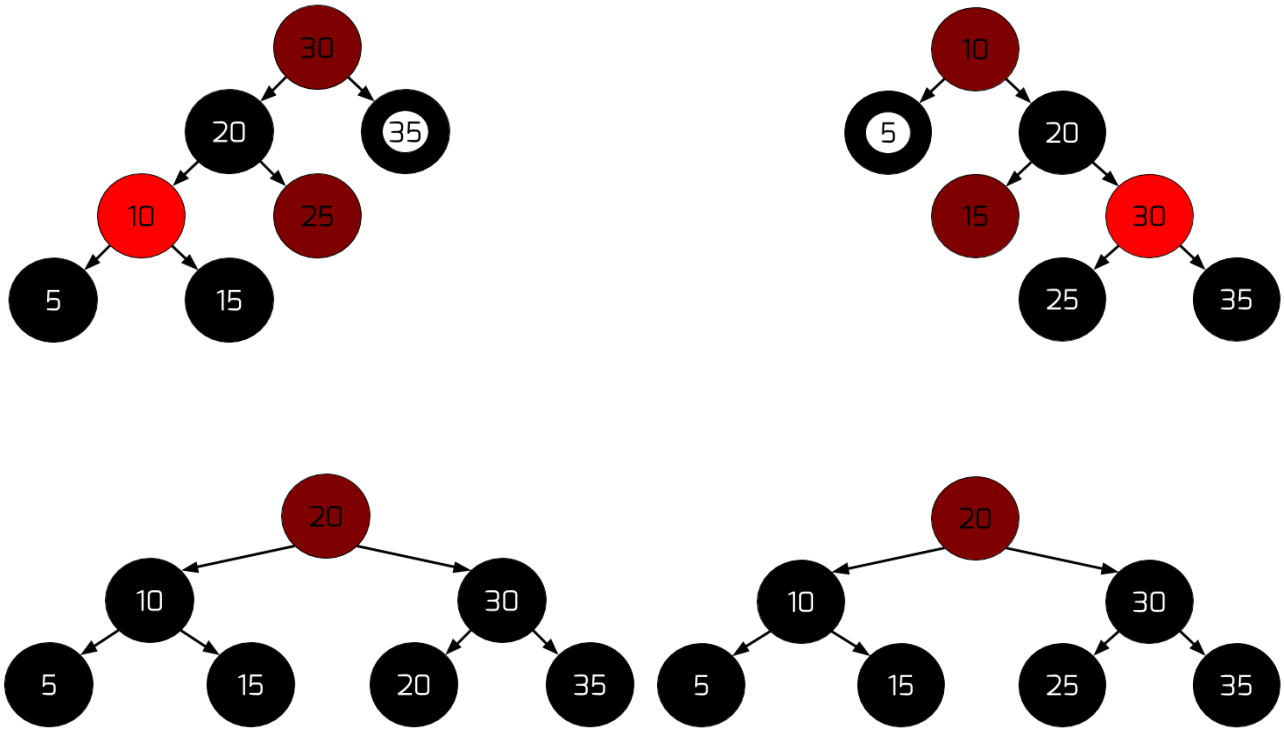

Figura 3.9: Cas 4 de la comprovació post-esborrat

Algorisme 3.7 Codi de la comprovació post-esborrat en C++

```
1  void RedBlack::comprovar_esborrar(RedBlackNode *n) {
2      RedBlackNode *germa;
3      Color tmp;
4      if(n->c == roig) n->c = negre;
5      else {
6          if(n == n->pare->esquerra) {
7              germa = n->pare->dreta;
8              if(germa->c == roig) { // Cas 1
9                  tmp = n->pare->c;
10                 n->pare->c = germa->c;
11                 germa->c = tmp;
12                 rotar_esquerra(n->pare);
13                 germa = n->pare->dreta;
14             }
15             if(germa->esquerra->c == negre and germa->dreta->c == negre) { // Cas
                   2
16                 germa->c = roig;
17                 comprovar_esborrar(n->pare);
18             } else if(germa->esquerra == roig) { // Cas 3
19                 germa->esquerra->c = n->pare->c;
20                 n->pare->c = negre;
21                 rotar_dreta(germa);
22                 rotar_esquerra(n->pare);
23             } else { // Cas 4
24                 germa->dreta->c = germa->c;
25                 germa->c = n->pare->c;
26                 n->pare->c = negre;
27                 rotar_esquerra(n->pare);
28             }
29         } else {
30             /* Mateix codi que en la branca anterior però
31             intercanviant les aparicions d' "esquerra" per "dreta"
32             i viceversa */
33         }
34     }
35 }
```

Capítol 4

Implementació

4.1 Estratègia de desenvolupament

La manera de dur a terme aquest desenvolupament és començar per obtenir una estructura operativa més bàsica i anar fent-la créixer fins a obtenir l'estructura desitjada. Per tant, la seqüència d'etapes per desenvolupar aquest projecte serà la següent:

1. Implementació de l'estructura de dades llista enllaçada.
2. Implementació d'una arbre binari de cerca no balancejat.
3. Implementació de l'arbre Red-Black.

Un avantatge d'aquesta aproximació és que permet des de l'inici mantenir la mateixa interfície d'entrades i sortides i, com a conseqüència, es pot re-aprofitar bona part del disseny ja fet en una etapa per a la posterior.

4.2 La llista enllaçada

4.2.1 Estructura

L'estructura i interfície de la llista enllaçada i que implementarem en aquesta primera etapa és la que correspon a la següent llista i que es pot veure escrita en el llenguatge VHDL en l'algorisme 4.1 a la pàgina següent.

- Com a senyals d'entrada apareixen les següents:
  - ordre: un bit per a indicar al disseny que té una petició de treball.
  - operació: un bit per a definir quin tipus de treball es demana.
    - si la senyal val 0, s'ha d'efectuar una cerca/inserció.
    - si la senyal val 1, s'ha d'efectuar una eliminació.
  - número: l'element a cercar, inserir o eliminar en la operació.
    - per a simplificar la implementació, es decideix que el tipus de l'element sigui un vector de 32 bits per tal de poder representar nombres enters.

  - rellotge: la senyal de rellotge del sistema, necessari per implementar aquest circuit seqüencial.

- Com a senyal de sortida tenim les següents:
  - fi: un bit per indicar que l'operació ha finalitzat.
  - resultat: un nombre que indica l'adreça on s'ha, o estava, desat el nombre o l'últim node visitat en cas de no trobar-lo.
  - trobat: un bit que indica si el nombre s'ha trobat o no.

Algorisme 4.1 Definició del mòdul de la llista enllaçada en VHDL

```
entity llista is
port ( rellotge : in std_logic;
       ordre : in std_logic;
       operacio : in std_logic;
       nombre : in std_logic_vector (31 downto 0);
       fi: out std_logic;
       trobat : out std_logic;
       resultat: out std_logic_vector(7 downto 0)
);
end llista;
```

Dintre d'aquest mòdul, guardarem els nodes en un vector de 256 posicions, anomenada *memòria*, per tal de utilitzar 8 bits com a adreça de memòria. Cada node, com es pot veure en l'algorisme 4.1, conté el valor a emmagatzemar i un apuntador al següent element de la llista.

Algorisme 4.2 Definició d'un node de la llista enllaçada

```
type node is record
    numero : std_logic_vector(31 downto 0);
    seguent : std_logic_vector(7 downto 0);
end record node;
```

En tot moment es pot veure el contingut de la memòria com a dues llistes:

1. Una llista de nodes lliures, que començarà per la posició apuntada per una variable anomenada *primer_element_lliure,* fins arribar a un node on el seu apuntador *següent* apunti a ell mateix. El camp *número* de cada node que formi part d'aquesta llista contindrà el valor numèric 0. El nombre d'elements presents en aquesta llista es mantindrà a la variable d'estat *elements_lliures.*

2. Una llista de nodes vàlids, amb els valors presents en aquesta llista. Aquesta llista començara pel node apuntat per la variable d'estat *primer_element_ocupat* i es navega per la llista seguint els valors del camp *següent* de cada node fins arribar a un node on el valor del seu camp *següent* sigui el mateix node.

4.2.2 Operacions

L'estructura bàsica de funcionament d'aquesta, i les següents, és la màquina d'estats. En una màquina d'estats, el circuit o algorisme, va avançant pas a pas a cada cicle. Com es pot veure en l'algorisme 4.3, en cada cicle executarà un pas que realitzarà modificacions sobre les variables d'estat del disseny i/o les senyal d'entrada/-sortida i en funció d'aquestes indicarà quin és el següent estat a avaluar en el següent cicle de rellotge.

Habitualment, i com serà en aquest projecte, l'estat actual es desarà en una variable d'estat i els valors possibles s'especificaran en una enumeració per tal de fer el codi més entenidor a ulls d'hom.

Algorisme 4.3 Estructura d'una màquina d'estats en VHDL

```
1  if rellotge'event and rellotge = '1' then
2      -- S'evaluarà un estat en cada cicle
3      case estat is -- estat és la variable que conté l'estat a evaluar
4          when Estat1 =>
5              -- codi a evaluar en aquest estat
6          when Estat2 =>
7              -- codi a evaluar en aquest estat
8          -- ...
9      end case;
10 end if;
```

4.2.2.1 Cerca

Com es mostra en l'algorisme 4.4 a la pàgina 41, tota cerca operació s'iniciarà estant en un estat inactiu o ociós. En aquest estat, si es detecta una nova sol·licitut d'una operació, passarà a un estat d'inici. En aquest segon estat, discriminarà si la llista està buida o no.

En cas que la llista estigui buida; si la operació que s'indica és una inserció en el següent cicle de rellotge anirem a l'estat d'*inserció* per fer efectiva l'operació. En cas que se'ns demani un esborrat, indiquem que s'ha finalitzat l'operació posant el valor *1* a la senyal *fi*, el valor *0* a la senyal *trobat* i el valor de la variable *primer_element_lliure* a la senyal *resultat.* En cas que la llista no estigui buida, anirem a l'estat de cerca i assignarem una variable iteradora la posició del primer element de la llista.

En l'estat de cerca, compararem el valor del camp *número* amb el valor proporcionat a través de les senyals d'entrada. En cas que els valors siguin iguals, indicarem que s'ha trobat l'element posant el valor *1* a la senyal *trobat* i assignarem aquesta posició de la memòria *,* i si es demanava realitzar una inserció, indicarem la finalització de l'operació ja que, com a conjunt que implementa aquesta llista, no s'admeten valors duplicats. En cas que es demanés realitzar un esborrat saltarem a l'estat d'esborrat per tal de fer efectiva l'operació.

En cas que els valors siguin diferents, es consulta el valor del camp *següent.* En cas que el seu valor sigui diferent al valor de la seva posició copia aquest valor a l'iterador per tal que en el següent cicle, que ens mantindrem en el mateix estat, consultarem aquesta posició. En cas que el camp *següent* faci referència a la mateixa posició de memòria en la qual ens trobem ens trobem al final de la llista. donat aquest cas, si es demana realitzar una inserció, en el següent cicle de rellotge, anirem a l'estat d'*inserció.* En cas contrari, indiquem la fi de l'operació.

4.2.2.2 Inserció

La inserció d'un element nou a la llista és ben senzilla, com es pot comprovar llegint l'algorisme 4.5 a la pàgina 42. Es tracta d'emmagatzemar un node en la posició apuntada per *primer_element_lliure* que contingui el numero que se'ns proporciona i, com que sempre afegim els nodes pel final, fem que el camp *següent* apunti a ell mateix. A partir d'ara, el punter *primer_element_lliure* fa referència el node que apuntava amb el camp *següent* el node utilitzat en la inserció. D'altra banda gestionem el punter *primer_element_ocupat* en cas que afegim el primer element a la llista, es decrementa el comptadors d'elements disponibles i indiquem que s'ha finalitzat l'operació i en quina posició hem inserit l'element.

4.2.2.3 Esborrat

L'esborrat d'un element de la llista, que es pot veure en l'algorisme 4.6 a la pàgina 42, es conforma de dues parts. La primera, i més òbvia, és l'esborrat del propi node, que consisteix en sobre escriure el camp *numero* pel valor 0 i segon per indicar que passa a apuntar en el camp *següent* a la posició apuntada per la variable *primer_element_lliure* i que aquesta variable apunti a aquest mateix node i s'augmenta en una unitat el comptador d'elements lliures. En la segona part s'han de gestionar una serie de punters per tal de mantenir l'estructura de la llista intacta. En aquesta part, s'han de contemplar 3 possibles casos:

1. L'element a esborrar sigui el primer de la llista. En aquest cas la variable primer_element_ocupat ha d'apuntar el següent node de la llista.
2. L'element a esborrar sigui l'últim de la llista. En aquest cas el node anterior de la llista s'ha d'apuntar a ell mateix indicant que és el nou últim node de la llista.
3. L'element a esborrar es trobi enmig de la llista. Aquest cas és una combinació dels dos, ja que l'anterior node ha d'apuntar al següent node de la llista.

I per finalitzar l'operativa, indiquem la fi de la mateixa amb la senyal *fi* i indiquem en quina posició es trobava l'element esborrat amb la senyal *resultat*.

4.3 L'arbre binari

4.3.1 Estructura

Es manté la mateixa interfície que es va implementar en el disseny de la llista enllaçada, tal i com es pot veure en l'algorisme 4.7 a la pàgina 43. Es justifica en el fet que totes dues estructures implementen la mateixa operativa, que és la del conjunt.

En canvi l'estructura dels nodes si que es modifica per adequar-la a les tipologia d'un node d'un arbre. Com es pot veure en l'algorisme 4.8 a la pàgina 43, es substitueix el punter següent pels punters anomenats esquerra i dreta. El punter anomenat dreta portarà la funció de punter al següent node buit tal i com feia el punter següent en els nodes de la llista enllaçada.

4.3.2 Operacions

4.3.2.1 Cerca

Es modifica l'estat de cerca es modifica, com es pot veure en 4.9 a la pàgina 44, per adequar-lo al mètode cerca d'un arbre binari de cerca. Concretament es modifica en els següents punts:

1. En cas que s'hagi trobat l'element i l'operació sigui un esborrat:
   (a) En cas que s'ha trobat l'element i l'operació sigui un esborrat i el node trobat tingui un o cap sub-arbre, es modifica el node pare en conseqüència. És a dir, el node pare s'apuntarà a ell mateix en cas que el node a eliminar sigui un node fulla o apuntarà al seu únic fill en cas que el node a esborrar tingui un fill.
   (b) En cas que el node a esborrar tingui dos sub-arbres; es buscarà el node de mínim valor del sub-arbre dret[1], que n'ocuparà el seu lloc.
2. En cas que s'hagi de navegar cap a un altre node, es consultarà el punter *esquerra* o *dreta* en funció de si el valor a cercar és menor o major al trobat en el node present, com es pot veure en l'algorisme 4.10 a la pàgina 45.
3. En cas que calgui inserir un nou node, inicialitzem l'apuntador *esquerra* o *dreta* segons sigui el cas, amb el valor de la variable global *primer_element_lliure*.

4.3.2.2 Inserció

En l'estat d'Inserció, l'únic canvi present es troba en la inicialització del nou node, ja que la seva estructura s'ha vist modificada. Com es pot veure en l'algorisme 4.11 a la pàgina 46, en concret s'inicialitzen els apuntadors *esquerra* i *dreta*, que s'apuntaran al seu propi node.

4.3.2.3 Esborrat

En el cas de portar a terme un esborrat, cal tenir present un pas previ que és la cerca del node; també dit el node *promocionat*; que re-emplaçarà el node a esborrar en cas que aquest últim tingui dos sub-arbres. Es comença la cerca d'aquest node començant des del fill dret del node a esborrar i anirem navegant pels successius fills esquerres fins que no es pugui avançar més. Llavors s'han de realitzar les següents accions, com es pot veure en l'algorisme 4.12 a la pàgina 47:

1. Si el node a promocionar té com a pare un node diferent al node a esborrar, ha de passar a apuntar amb el seu camp *esquerra*, al fill dret del node promocionat, en cas que en tingui, o ha d'apuntar-se a ell mateix.
2. El node pare del node a esborrar ha d'apuntar al node promocionat.
3. El node promocionat ha d'apuntar com a fill esquerra al fill esquerra del node a esborrar.
4. El node promocionat ha d'apuntar com a fill dret al fill dret del node a esborrar.
   (a) Un cas particular d'aquest fet té lloc quan el fill dret del node a esborrar no té fill esquerra i, conseqüentment, és el node a promocionar. En aquest cas, no s'ha de modificar el valor del camp dret del node promocionat.

[1] Es pot buscar també el node de màxim valor del sub-arbre esquerre, que és el cas simètric i equivalent.

4.4 L'arbre Red-Black

4.4.1 Estructura

Tal i com tenia lloc amb l'arbre binari de cerca, en l'algorisme 4.13 a la pàgina 48, es manté la mateixa interfície de senyals dels dos casos anteriors.

Com es pot comprovar en l'algorisme 4.14 a la pàgina 48, en l'estructura dels nodes es realitzen modificacions. S'afegeix un camp d'un bit per indicar el color del node i s'afegeix una apuntador addicional que servirà per apuntar al node pare del node en qüestió. Per tal de fer el codi més llegible es creen dos constants per definir els dos colors possibles d'un node.

Un altre canvi que afecta als nodes de l'arbre és en els possibles valors dels apuntadors. Fins ara, quan un apuntador no feia referència a cap altre node, es feia que apuntés al mateix node que pertanyia l'apuntador. Ara tots els apuntadors que no apuntin a cap node apuntaran a la posició 255.

En la inicialització de la llista de nodes, el color del node en la posició 255 serà sempre negre.

4.4.2 Operacions

4.4.2.1 Cerca

Es modifica la part de l'estat de cerca on s'ha trobat l'element sol·licitat en els següents punts tal i com es mostra en l'algorisme 4.15 a la pàgina 49:

1. Per consultar si un dels apuntadors esquerra o dreta apunta a una altre node, ara es comprovarà si el valor és diferent al valor 255, *FF* en hexadecimal.
2. En cas d'esborrat:
   (a) Si el node a esborrar té un fill, aquest últim ha d'apuntar al node a esborrar mitjançant l'apuntador *pare.*
   (b) Es memoritza el color del node a esborrar.
   (c) En cas que el node a esborrar no tingui cap fill, es crea una *fulla fantasma,* que apuntarà al pare del node a esborrar per tal de poder fer les comprovacions post-esborrat pertinents.

En l'algorisme 4.16 a la pàgina 50 es veu que en la part de navegació cap al següent node de l'arbre, es modifica per tal de memoritzar la posició del node germà del pròxim node a visitar i ens guardem en una variable si el pròxim node a visitar és el fill dret o no.

4.4.2.2 Inserció

El codi de l'estat d'inserció de l'algorisme 4.17 a la pàgina 51 es modifica per tal de satisfer els següents punts:

1. El canvi dels valor dels apuntadors esquerra i dreta que ara contindran el valor 255, ja que el nou node sempre serà un node terminal.
2. S'afegeix codi per assignar el color del nou node en funció de si el nou node és l'arrel de l'arbre. En cas afirmatiu, el color serà el negre seguint la regla 2 dels invariants de l'arbre Red-Black, en cas contrari serà de color negre el nou node.

3. S'afegeix codi per assignar el valor adequat per l'apuntador al node pare, en funció de si el nou node és l'arrel de l'arbre o no. Si el nou node és l'arrel, l'apuntador contindrà el valor 255, indicant que no ha cap node per sobre d'ell mateix. En cas contrari, contindrà la posició de l'últim node visitat en la cerca.

4. Finalment, si l'últim node visitat era un node roig, i per tant es viola la regla 4 dels invariants, anirem en el següent cicle de rellotge a l'estat de comprovació post-inserció; per tal de corregir aquest conflicte.

4.4.2.3 Comprovació post-inserció

El cas dels germans rojos Per tal de resoldre aquest cas, en l'algorisme 4.18 a la pàgina 51, cal canviar el color dels 3 nodes, pare, germà i el seu pare i re-apuntar als nodes corresponents tenint present que *pugem* dos nodes en l'arbre tal i com s'ha exposat en el punt 3.4.2.1 a la pàgina 18. Un cop fetes les re-assignacions, s'indica que hem de tornar a començar les comprovacions indicant que el proper estat a visitar és l'estat actual tal i com es pot veure en l'algorisme 4.19 a la pàgina 52.

El cas dels 3 nodes en línia En la resolució d'aquest cas, i aprofitant el fet que els canvis no es faran efectius fins al següent cicle de rellotge, en l'algorisme 4.20 a la pàgina 52 es realitza la rotació, *in situ* i el canvi de colors seguint els passos exposats al punt 3.4.2.2 a la pàgina 20. Com que no s'ha de realitzar cap comprovació ni acció addicional, indiquem que el següent estat és el de finalització de les comprovacions, per tal que pugui indicar que l'operació ha finalitzat en el següent cicle.

Es mostra el cas en que els tres nodes es troben alineats cap a la dreta. L'altre cas és idèntic per substituint esquerra per dreta i viceversa.

El cas de la ziga-zaga En la resolució del cas de la ziga-zaga es resol en l'algorisme 4.21 a la pàgina 53 duent a terme les rotacions segons l'exposat en el punt 3.4.2.3 a la pàgina 21 *in situ* i fent els canvis de color corresponents en el mateix cicle. Com que no s'han de realitzar cap altra comprovació ni acció addicional indiquem que el següent estat és el de finalització de les comprovacions.

L'altre possibilitat d'aquest cas és simètric, només cal canviar esquerra per dreta i viceversa.

4.4.2.4 Esborrat

Es modifica l'estat de promocionar per tal de satisfer els següents punts tal i com es pot veure en l'algorisme 4.22 a la pàgina 54:

1. Per comprovar si un apuntador apunta a un node o no, es canvia el valor a comparar pel valor 255, FF en hexadecimal.

2. S'afegeix codi per gestionar el punter pare per tal que el node fill del node a promocionar apunti al seu nou node pare.

3. Es memoritza des de quin node s'han de començar a fer les comprovacions post-esborrat.

   (a) En cas que es requereixi, es crea una *fulla fantasma* en el node amb adreça 255.

4. Es memoritza el color del node a promocionar, que serà el que s'emprarà per indicar si s'han de realitzar les comprovacions pertinents o no.

5. S'afegeix codi per tal que el node a promocionar es quedi amb el color i la referència al pare del node a esborrar.

En l'algorisme 4.23 a la pàgina 55, es modifica l'estat d'esborrar en els següents punts:

1. En l'esborrat del node, s'afegeixen valors pels nous camps de color i apuntador al pare.
2. S'afegeix inicialització de apuntadors pels cas que s'hagi de realitzar les comprovacions post-esborrat.

4.4.2.5 Comprovació post-esborrat

El cas del germà roig En l'algorisme 4.24 a la pàgina 56 es realitza la rotació i el canvi de color *in situ* seguint els criteris exposats en el punt 3.4.3.1 a la pàgina 22. Acte seguit s'assignen nou valors a les variables apuntadores seguint els nou canvis en l'estructura de l'arbre i s'indica que en el següent cicle es comprovaran la resta de casos.

El cas dels nebots negres Per a la resolució d'aquest cas, es procedeix en l'algorisme 4.25 a la pàgina 56 al canvi de color del node i a escalar una posició en l'arbre per tornar a començar amb les comprovacions tal i com s'indicava en el punt 3.4.3.2 a la pàgina 23.

El cas del nebot pròxim roig Per tal de resoldre aquesta situació, en l'algorisme 4.26 a la pàgina 57 es procedeix a realitzar la doble rotació i el canvi de color exposat en el punt 3.4.3.3 a la pàgina 23. El cas en que el nebot pròxim roig és el fill esquerra, i per tant el node actual és el fill esquerra també és simètric al cas exposat en l'algorisme 4.26 a la pàgina 57.

El cas del nebot llunyà roig Per tal de resoldre aquesta situació, en l'algorisme 4.27 a la pàgina 58 es realitza la rotació i el canvi de colors tal i com s'ha exposat en el punt 3.4.3.4 a la pàgina 23. En el cas que el nebot llunyà fos el fill dret, i per tant el node actual és el fill esquerra és simètric a l'algorisme 4.27 a la pàgina 58.

Algorisme 4.4 Cerca d'un element a la llista en VHDL

```
when Inici =>
    if unsigned(elements_lliures) = 255 then -- La llista està buida
        if operacio = '0' then
            e <= Insercio;
        else
            fi <= '1';
            e <= Ocios;
            resultat <= x"00";
        end if;
    else
        e <= Cerca;
        punter <= primer_element_ocupat;
    end if;
when Cerca =>
    if llista(conv_integer(punter)).numero = numero then -- Trobat
        trobat <= '1';
        if operacio = '1' then e <= Esborrat;
        else
            fi <= '1';
            e <= Ocios;
            resultat <= punter;
        end if;
    else
        if llista(conv_integer(punter)).seguent /= punter then
            -- Continuem cercant
            anterior <= punter;
            punter <= llista(conv_integer(punter)).seguent;
        else -- Fina de trajecte
            if operacio = '0' and unsigned(elements_liures) > 0 then
                e <= Insercio;
                llista(conv_integer(punter)).seguent <= primer_element_lliure;
            else
                fi <= '1';
                resultat <= punter;
                e <= Ocios;
            end if;
        end if;
    end if;
```

Algorisme 4.5 Inserció d'un element a la llista en VHDL

```
when Insercio =>
    fi <= '1';
    resultat <= primer_element_lliure;
    llista(conv_integer(primer_element_lliure)) <= (numero => numero,
       seguent => primer_element_lliure);
    primer_element_liure <= llista(conv_integer(primer_element_lliure)).seguent;
    if unsigned(primer_elemento_ocupado) = 255 then
        primer_elemento_ocupat <= primer_elemento_lliure;
    end if;
    elements_lliures <= elements_lliures - 1;
    e <= Ocios;
```

Algorisme 4.6 Esborrat d'un element de llista en VHDL

```
when Esborrat =>
    fi <= '1';
    resultat <= punter;
    llista(conv_integer(punter)) <= (numero => X"00000000", seguent =>
       primer_element_lliure);
    primer_element_lliure <= punter;
    if llista(conv_integer(punter)).seguent = punter then
        if punter /= primer_element_ocupat then
            llista(conv_integer(anterior)).seguent <= anterior;
        end if;
    else
        if punter /= primer_element_ocupat then
            llista(conv_integer(anterior)).seguent <= llista(conv_integer(punter)
               ).seguent;
        end if;
    end if;
    if punter  = primer_element_ocupat then
        primer_element_ocupat <= llista(conv_integer(punter)).seguent;
    end if;
    elements_lliures <= elements_lliures + 1;
    e <= Ocios;
```

Algorisme 4.7 Definició del mòdul de l'arbre binari de cerca

```
entity arbre_binari is
    port (rellotge : in std_logic;
        ordre : in std_logic;
        operacio : in std_logic;
        numero : in std_logic_vector (31 downto 0);
        fi: out std_logic;
        trobat: out std_logic;
        resultat: out std_logic_vector(7 downto 0)
);
end arbre_binari;
```

Algorisme 4.8 Definició d'un node de l'arbre binari de cerca

```
type node is record
    numero : std_logic_vector (31 downto 0);
    esquerra : std_logic_vector (7 downto 0);
    dreta : std_logic_vector (7 downto 0);
end record;
```

Algorisme 4.9 Cerca d'un element en un arbre binari

```
when Cerca =>
if llista(conv_integer(punter)).numero = numero then -- Trobat
    trobat <= '1';
    if operacio = '1' then
        if llista(conv_integer(punter)).esquerra = punter and llista(conv_integer
           (punter)).dreta = punter then
        -- Cas d'esborrar un node fulla
        if anterior /= puntero then
            if numero < llista(conv_integer(anterior)).numero then
                llista(conv_integer(anterior)).esquerra <= anterior;
            else
                llista(conv_integer(anterior)).dreta <= anterior;
            end if;
        end if;
        e <= Esborrat;
        elsif llista(conv_integer(punter)).esquerra = punter then
            -- Cas d' esborrat d'un node amb un únic sub-arbre
            if anterior /= punter then
                if numero < llista(conv_integer(anterior)).numero then
             llista(conv_integer(anterior)).esquerra <= llista(conv_integer(
                punter)).dreta;
                else
                 llista(conv_integer(anterior)).dreta <= llista(conv_integer(
                    punter)).dreta;
                end if;
            else
            primer_element_ocupat <= llista(conv_integer(punter)).dreta;
            end if;
            e <= Esborrat;
        elsif llista(conv_integer(punter)).dret = punter then
                -- Cas simètric a l'anterior
        else
            -- El node a esborrar te 2 sub-arbres
            substitut <= llista(conv_integer(punter)).dreta;
            pare_substitut <= punter;
            e <= Promocionar;
            end if;
        else
            fin <= '1';
            e <= Ocios;
            resultat <= punter;
        end if;
```

Algorisme 4.10 Navegació cap al següent node en una arbre binari de cerca

```
else
    -- Cal anar a l'esquerra?
    if unsigned(numero) < unsigned(llista(conv_integer(punter)).numero) then
        -- Hi ha node a visitar?
            if llista(conv_integer(punter)).esquerra /= punter then
                anterior <= punter;
                punter <= llista(conv_integer(punter)).esquerra;
            else
                if operacio = '0' and unsigned(elements_lliures) > 0 then
                    e <= Insercio;
                    llista(conv_integer(punter)).esquerra <=
                        primer_element_lliure;
                else
                    fi <= '1';
                    resultat <= punter;
                    e <= Ocios;
                end if;
            end if;
    -- Cal anar a la dreta?
    elsif unsigned(numero) > unsigned(llista(conv_integer(punter)).numero) then
        -- Hi ha node a visitar?
        if llista(conv_integer(punter)).dreta /= punter then
            anterior <= punter;
            punter <= llista(conv_integer(punter)).dreta;
        else
            if operacio = '0' and unsigned(elements_lliures) > 0 then
                e <= Insercio;
                llista(conv_integer(punter)).dreta <= primer_elemento_lliure;
            else
                fi <= '1';
                resultat <= punter;
                e <= Ocios;
            end if;
        end if;
    else null;
    end if;
end if;
```

Algorisme 4.11 Inserció d'un element a l'arbre binari en VHDL

```
when Insercio =>
    fi <= '1';
    resultat <= primer_element_lliure;
    llista(conv_integer(primer_element_lliure)) <= (numero => numero, esquerra =>
        primer_element_lliure, dreta => primer_element_liure);
    primer_element_lliure <= llista(conv_integer(primer_element_lliure)).dreta;
    if elements_lliures = x"ff" then
        primer_element_ocupat <= primer_element_lliure;
    end if;
    elements_lliures <= elements_lliures - 1;
    e <= Ocios;
```

Algorisme 4.12 Esborrat d'un element de l'arbre binari

```
when Promocionar =>
    -- Busquem el valor + petit del sub-arbre
    if llista(conv_integer(substitut)).esquerra /= substitut then
        pare_substitut <= substitut;
        substitut <= llista(conv_integer(substitut)).esquerra;
        e <= Promocionar;
    else
        -- Trobat l'element més petit
        if padre_substitut /= punter then
            -- Trenquem l'enllaç amb el node a promocionar
            if llista(conv_integer(substitut)).dreta = substitut then
                llista(conv_integer(pare_substitut)).esquerra <= pare_substitut;
            else
                llista(conv_integer(pare_substitut)).esquerra <= llista(
                    conv_integer(substitut)).dreta;
            end if;
            -- Substituïm el node
                llista(conv_integer(substitut)).dreta <= llista(conv_integer(
                    punter)).dreta;
                llista(conv_integer(substitut)).esquerra <= llista(conv_integer(
                    punter)).esquerra;
        else
            llista(conv_integer(substitut)).esquerra <= llista(conv_integer(
                punter)).esquerra;
        end if;
        -- Afegim el node promocionat al node anterior al node a esborrar.
        if anterior = punter then
            primer_element_ocupat <= substitut;
        else
            if punter = llista(conv_integer(anterior)).esquerra then
                llista(conv_integer(anterior)).esquerra <= substitut;
            else
                llista(conv_integer(anterior)).dreta <= substitut;
            end if;
        end if;
        e <= Esborrat;
    end if;
when Esborrat =>
    fi <= '1';
    resultat <= punter;
    llista(conv_integer(punter)) <= (numero => X"00000000", izquierda => punter,
        derecha => primer_element_lliure);
    primer_element_lliure <= punter;
    elements_lliures <= elements_lliures + 1;
    e <= Ocios;
```

Algorisme 4.13 Definició del mòdul de l'arbre *Red-Black*

```
entity redblack is
    port (rellotge : in std_logic;
        ordre : in std_logic;
        operacio : in std_logic;
        numero : in std_logic_vector (31 downto 0);
        fi: out std_logic;
        trobat: out std_logic;
        resultat: out std_logic_vector(7 downto 0)
);
end redblack;
```

Algorisme 4.14 Estructura d'un node de l'arbre *Red-Black*

```
constant negre : std_logic := '1';
constant roig : std_logic := '0';
type node is record
    c : std_logic;
    pare : std_logic_vector(7 downto 0);
    numero : std_logic_vector(31 downto 0);
    esquerra : std_logic_vector(7 downto 0);
    dreta : std_logic_vector (7 downto 0);
end record;
```

Algorisme 4.15 Cerca d'un element a l'arbre *Red-Black*

```
when cerca =>
    if llista(conv_integer(punter)).numero = numero then
        trobat <= '1';
        if operacio = '1' then
            color_vell <= llista(conv_integer(punter)).c;
            if llista(conv_integer(punter)).esquerra = x"ff" and llista(
               conv_integer(punter)).dreta = x"ff" then
                if primer_element_ocupat /= punter then
                    if numero < llista(conv_integer(anterior)).numero then
                        llista(conv_integer(anterior)).esquerra <= x"ff";
                    else
                        llista(conv_integer(anterior)).dreta <= x"ff";
                    end if;
                end if;
                -- creem 'fulla fantasma'
                llista(255).pare <= llista(conv_integer(punter)).pare;
                tmp_punter <= x"ff"; e <= esborrat;
            elsif llista(conv_integer(punter)).esquerra = x"ff" then
                if primer_element_ocupat /= punter then
                    if numero < llista(conv_integer(anterior)).numero then
                        llista(conv_integer(anterior)).esquerra <= llista(
                           conv_integer(punter)).dreta;
                    else
                        llista(conv_integer(anterior)).dreta <= llista(
                           conv_integer(punter)).dreta;
                    end if;
                    llista(conv_integer(lista(conv_integer(punter)).dreta)).pare
                       <= anterior;
                else
                    primer_element_ocupat <= llista(conv_integer(punter)).dreta;
                    llista(conv_integer(lista(conv_integer(punter)).dreta)).pare
                       <= x"ff";
                end if;
                e <= esborrat;
                tmp_puntero <= llista(conv_integer(punter)).dreta;
            elsif llista(conv_integer(punter)).dreta = x"ff" then
                -- cas simètric a l'anterior
            else
                substitut <= llista(conv_integer(punter)).dreta;
                pare_substitut <= punter; e <= promocionar;
            end if;
        else
            fi <= '1'; e <= ocios; resultat <= punter;
        end if;
```

Algorisme 4.16 Navegació cap al següent node en un arbre *Red-Black*

```
    else
        -- Anem cap a l'esquerra?
        if unsigned(numero) < unsigned(lista(conv_integer(puntero)).numero) then
            -- Hi ha node a visitar?
            if llista(conv_integer(punter)).esquerra /= x"ff" then
                superior <= anterior;
                anterior <= punter;
                germa <= llista(conv_integer(punter)).dreta;
                punter <= llista(conv_integer(punter)).esquerra;
                p_der <= false;
            else
                if operacio = '0' and unsigned(elementos_libres) > 0 then
                    e <= insercio;
                    llista(conv_integer(punter)).esquerra <=
                        primer_element_lliure;
                else
                    fi <= '1';
                    resultat <= punter;
                    e <= ocios;
                end if;
            end if;
        -- Anem cap a la dreta?
        elsif unsigned(numero) > unsigned(lista(conv_integer(punter)).numero)
            then
            -- Hi ha node a visitar?
            if llista(conv_integer(punter)).dreta /= x"ff" then
                superior <= anterior;
                anterior <= punter;
                punter <= llista(conv_integer(punter)).dreta;
                germa <= llista(conv_integer(punter)).esquerra;
                p_der <= true;
            else
                if operacio = '0' and unsigned(elementos_libres) > 0 then
                    e <= insercio;
                    llista(conv_integer(punter)).dreta <= primer_element_lliure;
                else
                    fi <= '1';
                    resultat <= punter;
                    e <= ocios;
                end if;
            end if;
        else null;
        end if;
    end if;
```

Algorisme 4.17 Inserció d'un element en un arbre *Red-Black*

```
when insercio =>
    resultat <= primer_elemento_libre;
    nou <= primer_element_lliure;
    llista(conv_integer(primer_element_lliure)).numero <= numero;
    llista(conv_integer(primer_element_lliure)).esquerra <= x"ff";
    llista(conv_integer(primer_element_lliure)).dreta <= x"ff";
    primer_element_lliure <= llista(conv_integer(primer_element_lliure)).dreta;
    if elements_lliures = x"ff" then
        primer_element_ocupat <= primer_element_lliure;
        llista(conv_integer(primer_element_lliure)).c <= negre;
        llista(conv_integer(primer_element_lliure)).pare <= x"ff";
        e <= ocios;
        fi <= '1';
    else
        llista(conv_integer(primer_element_lliure)).pare <= punter;
        llista(conv_integer(primer_element_lliure)).c <= roig;
        if llista(conv_integer(punter)).c = roig then
            e <= ci;
        else
            fi <= '1';
            e <= ocios;
        end if;
    end if;
elements_lliures <= elements_lliures - 1;
```

Algorisme 4.18 Resolució del cas dels germans rojos

```
-- El cas dels germans rojos
llista(conv_integer(germa)).c <= negre;
llista(conv_integer(punter)).c <= negre;
llista(conv_integer(anterior)).c <= roig;
nou <= anterior;
puntero <= superior;
anterior <= llista(conv_integer(superior)).pare;
e <= ciescalada;
```

Algorisme 4.19 Resolució del cas dels germans rojos

```
when ciescalada =>
superior <= llista(conv_integer(anterior)).pare;
if llista(conv_integer(anterior)).esquerra = punter then
    germa <= llista(conv_integer(anterior)).dreta;
    p_der <= false;
else
    germa <= llista(conv_integer(anterior)).esquerra;
    p_der <= true;
end if;
e <= ci;
```

Algorisme 4.20 Resolució del cas dels 3 nodes en línia

```
-- El cas dels 3 nodes en línia (a la dreta)
if primer_element_ocupat = anterior then
    primer_element_ocupat <= punter;
elsif llista(conv_integer(superior)).esquerra = anterior then
    llista(conv_integer(superior)).esquerra <= punter;
else
    llista(conv_integer(superior)).dreta <= punter;
end if;
llista(conv_integer(punter)) <= (c => negre, pare => llista(conv_integer(anterior
    )).pare,  numero => llista(conv_integer(punter)).numero, esquerra => anterior,
     dreta => nou);
llista(conv_integer(anterior)) <= (c => roig, pare => punter, numero => llista(
    conv_integer(anterior)).numero, esquerra => germa, dreta => llista(
    conv_integer(punter)).esquerra);
if llista(conv_integer(punter)).esquerra /= x"ff" then
llista(conv_integer(llista(conv_integer(punter)).esquerra)).pare <= anterior;
end if;
e <= finci;
```

Algorisme 4.21 Resolució del cas de la ziga-zaga

```
-- El cas de la ziga-zaga
if primer_element_ocupat = anterior then
    primer_element_ocupat <= nou;
elsif llista(conv_integer(superior)).esquerra = anterior then
    llista(conv_integer(superior)).esquerra <= nou;
else
    llista(conv_integer(superior)).dreta <= nou;
end if;
llista(conv_integer(nou)) <= (c => negre, pare => llista(conv_integer(anterior)).
   pare, numero => llista(conv_integer(nou)).numero, esquerra => anterior, dreta
    => punter);
llista(conv_integer(anterior)) <= (c => roig, pare => nou, numero => llista(
   conv_integer(anterior)).numero, esquerra => germa, dreta => llista(
   conv_integer(nuevo)).esquerra);
llista(conv_integer(punter)) <= (c => roig, pare => nou, numero => llista(
   conv_integer(punter)).numero, esquerra => llista(conv_integer(nou)).dreta,
   dreta => llista(conv_integer(punter)).dreta);
if llista(conv_integer(nou)).esquerra /= x"ff" then
    llista(conv_integer(llista(conv_integer(nou)).esquerra)).pare <= anterior;
end if;
if llista(conv_integer(nou)).dreta /= x"ff" then
    llista(conv_integer(llista(conv_integer(nou)).dreta)).pare <= punter;
end if;
e <= finci;
```

Algorisme 4.22 Promoció d'un element de l'arbre *Red-Black*

```
when promocionar =>
    -- Cerquem l'element més petit del sub-arbre
    if llista(conv_integer(substitut)).esquerra /= x"ff" then
        pare_substitut <= substitut;
        substitut <= llista(conv_integer(substitut)).esquerra;
        e <= promocionar;
    else
        -- Trobat l'element més petit
        tmp_punter <= llista(conv_integer(substitut)).derecha;
        if pare_substitut /= punter then
            llista(conv_integer(pare_substitut)).esquerra <= llista(conv_integer(
                substitut)).dreta;
            if llista(conv_integer(substitut)).dreta /= x"ff" then
            llista(conv_integer(lista(conv_integer(substitut)).dreta)).pare <=
                pare_substitut;
            else
                llista(255).pare <= pare_substitut;
            end if;
            -- Re-emplacem el node
            llista(conv_integer(substitut)).dreta <= llista(conv_integer(punter))
                .dreta;
            llista(conv_integer(llista(conv_integer(punter)).dreta)).pare <=
                substitut;
        else
            llista(conv_integer(llista(conv_integer(substitut)).dreta)).pare <=
                substitut;
        end if;
        llista(conv_integer(substitut)).esquerra <= llista(conv_integer(punter)).
            esquerra;
        llista(conv_integer(lista(conv_integer(punter)).esquerra)).pare <=
            substitut;
        color_vell <= llista(conv_integer(substitut)).c;
        llista(conv_integer(substitut)).c <= llista(conv_integer(punter)).c;
        if primer_element_ocupat = punter then
            primer_element_ocupat <= substitut;
        else
            llista(conv_integer(substitut)).pare <= anterior;
            if punter = llista(conv_integer(anterior)).esquerra then
                llista(conv_integer(anterior)).esquerra <= substitut;
            else
                llista(conv_integer(anterior)).dreta <= substitut;
            end if;
        end if;
        e <= borrado;
    end if;
```

Algorisme 4.23 Esborrat d'un element de l'arbre *Red-Black*

```
when esborrat =>
    resultat <= punter;
    llista(conv_integer(punter)) <= (c => negre, pare => x"ff", numero => x"
       00000000", esquerra => punter, dreta => primer_element_lliure);
    primer_element_lliure <= punter;
    elements_lliures <= elements_lliures + 1;
    if color_vell = negre then
        e <= cb;
        punter <= tmp_punter;
        anterior <= llista(conv_integer(tmp_punter)).pare;
        superior <= llista(conv_integer(llista(conv_integer(tmp_punter)).pare)).
           pare;
        if tmp_punter = llista(conv_integer(lista(conv_integer(tmp_punter)).pare)
           ).esquerra then
            germa <= llista(conv_integer(llista(conv_integer(tmp_punter)).pare)).
               dreta;
            p_der <= false;
        else
            germa <= llista(conv_integer(llista(conv_integer(tmp_punter)).pare)).
               esquerra;
            p_der <= true;
        end if;
    else
        e <= ocios;
        fi <= '1';
    end if;
```

Algorisme 4.24 Resolució del cas del germà roig

```
-- El cas del germà roig (esquerra) roig
if anterior = primer_element_ocupat then
    primer_element_ocupat <= germa;
elsif llista(conv_integer(superior)).esquerra = anterior then
    llista(conv_integer(superior)).esquerra <= germa;
else
    llista(conv_integer(superior)).dreta <= germa;
end if;
llista(conv_integer(germa)).c <= negre;
llista(conv_integer(germa)).pare <= llista(conv_integer(anterior)).pare;
llista(conv_integer(germa)).dreta <= anterior;
llista(conv_integer(anterior)).c <= roig;
llista(conv_integer(anterior)).pare <= germa;
llista(conv_integer(anterior)).esquerra <= fill_dret;
if fill_dret /= x"ff" then
    llista(conv_integer(fill_dret)).pare <= anterior;
end if;
superior <= germa;
germa <= fill_dret;
fill_esquerra <= llista(conv_integer(fill_dret)).esquerra;
fill_dret <= llista(conv_integer(fill_dret)).dreta;
e <= cbc234;
```

Algorisme 4.25 Resolució del cas dels nebots negres

```
-- El cas dels nebots negres
llista(conv_integer(germa)).c <= roig;
punter <= anterior;
anterior <= superior;
superior <= llista(conv_integer(superior)).pare;
if anterior = llista(conv_integer(superior)).esquerra then
    germa <= llista(conv_integer(superior)).dreta;
    p_der <= false;
else
    germa <= llista(conv_integer(superior)).esquerra;
    p_der <= true;
end if;
e <= cb;
```

Algorisme 4.26 Resolució del cas del nebot pròxim roig

```
-- El cas del nebot pròxim (dret) roig
if anterior = primer_elemento_ocupat then
    primer_element_ocupat <= fill_dret;
elsif llista(conv_integer(superior)).esquerra = anterior then
    llista(conv_integer(superior)).esquerra <= fill_dret;
else
    llista(conv_integer(superior)).dreta <= fill_dret;
end if;
llista(conv_integer(fill_dret)).c <= llista(conv_integer(anterior)).c;
llista(conv_integer(fill_dret)).pare <= llista(conv_integer(anterior)).pare;
llista(conv_integer(fill_dret)).esquerra <= germa;
llista(conv_integer(fill_dret)).dreta <= anterior;
llista(conv_integer(germa)).pare <= fill_dret;
llista(conv_integer(germa)).dreta <= llista(conv_integer(fill_dret)).esquerra;
if llista(conv_integer(fill_dret)).esquerra /= x"ff" then
    llista(conv_integer(llista(conv_integer(fill_dret)).esquerra)).pare <= germa;
end if;
llista(conv_integer(anterior)).c <= negre;
llista(conv_integer(anterior)).pare <= fill_dret;
llista(conv_integer(anterior)).esquerra <= llista(conv_integer(fill_dret)).dreta;
if llista(conv_integer(fill_dret)).dreta /= x"ff" then
    llista(conv_integer(lista(conv_integer(fill_dret)).dreta)).pare <= anterior;
end if;
punter <= primer_element_ocupat;
e <= fincb;
```

Algorisme 4.27 Resolució del cas del nebot llunyà roig

```
-- El cas del nebot llunyà (esquerra) roig
if anterior = primer_element_ocupat then
    primer_element_ocupat <= germa;
elsif llista(conv_integer(superior)).esquerra = anterior then
    llista(conv_integer(superior)).esquerra <= germa;
else
    llista(conv_integer(superior)).dreta <= germa;
end if;
llista(conv_integer(germa)).c <= llista(conv_integer(anterior)).c;
llista(conv_integer(germa)).pare <= llista(conv_integer(anterior)).pare;
llista(conv_integer(germa)).dreta <= anterior;
llista(conv_integer(anterior)).c <= negre;
llista(conv_integer(anterior)).pare <= germa;
llista(conv_integer(anterior)).esquerra <= fill_dret;
llista(conv_integer(fill_esquerra)).c <= negre;
if fill_dret /= x"ff" then
    llista(conv_integer(fill_dret)).pare <= anterior;
end if;
punter <= primer_element_ocupat;
e <= fincb;
```

Capítol 5

Desenvolupament

5.1 Entorn de desenvolupament

Tenint present que la tarja objectiu on els responsables del projecte pretenen fer funcionar el disseny sol·licitat és una placa model Zedboard de la casa Avnet i que aquesta conté una FPGA model Zynq 7020 del fabricant Xilinx; s'utilitzarà l'entorn de desenvolupament proporcionat per aquest últim en la seva plana web. L'entorn o IDE en qüestió es diu Vivado Desing Suite i la seva versió gratuïta Vivado Webpack permet el treball en aquesta placa i és, en conseqüència, l'opció escollida per a la realització d'aquest projecte. Aquesta eina es pot descarregar des del següent enllaç[1]:

https://www.xilinx.com/support/download.html.

Aquesta eina es troba disponible tant per a Sistemes Operatius GNU/Linux com Windows. En aquest cas s'escull la versió per al sistema operatiu Windows ja que és el que es troba instal·lat en l'ordinador personal on es desenvoluparà aquest projecte.

5.2 Utilització de la *Block RAM*

5.2.1 Introducció

En el transcurs de la implementació de l'arbre, s'ha donat el cas que al sintetitzar el disseny, l'eina mostrava en el seus informes que el disseny era massa gran per encabir en el dispositiu. Tenint present que bona part de la capacitat del dispositiu s'emprava en una estructura de dades que és una memòria de dades; era desitjable trobar una manera de tenir aquesta estructura forma de la FPGA, tenint present a més que en la placa de la FPGA; es pot veure en la imatge del capítol d'introducció a les FPGA; conté memòria RAM en quantitat més que suficient per encabir-hi la memòria.

5.2.2 Definició del control·lador de BRAM

El passos a seguir per tal de poder utilitzar aquest mòduls de memòria es descriuen a continuació i agafant de referència la finestra de l'aplicatiu Vivado mostrada en la figura 5.1 a la pàgina següent:

[1] Recomano que en el moment d'instal·lar la eina com a l'actualitzar-la en una de les seves actualitzacions trimestrals es reservi temps i espai en el disc dur de l'ordinador ja que es tracta d'un procès que inclou la descàrrega d'uns 5 GiB d'informació aproximadament.

1. Obrir l'entorn Vivado de Xilinx i obrir en projecte en el qual estiguem treballant.
2. En el panell lateral *Flow Navigator*, cal sel·leccionar l'opció *IP Catalog*.
3. Apareixerà una finestra en la pantalla principal. En la llista d'opcions cal seleccionar l'opció *Block Memory Generator* seguint el camí *Vivado Repository* → *Basic Elements* → *Memory Elements* → *Block Memory Generator*.
   (a) O bé seguint el camí *Vivado Repository* → *Memories & Storage Elements* → *RAMs & ROMs & BRAM* → *Block Memory Generator*.

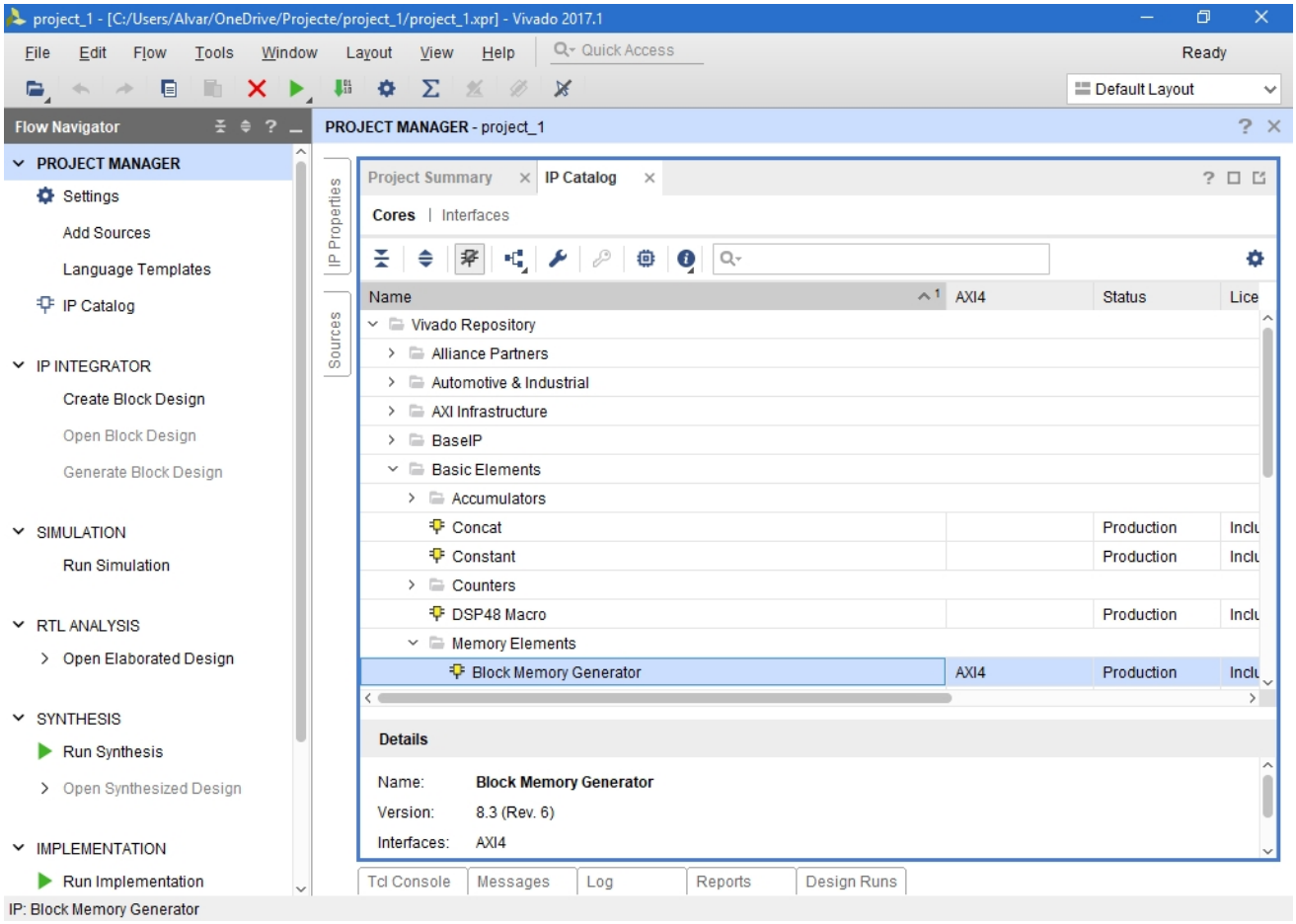

Figura 5.1: Localització del catàleg d'IPs i amb l'opció de *Block Memory Generator* activada

A continuació, apareixerà el següent quadre de diàleg que es mostra en la figura 5.2 a la pàgina següent, on podrem especificar el paràmetres per tal de model·lar el mòdul de memòria a les nostres necessitats.

El paràmetres que ens interessen especificar són els següents:

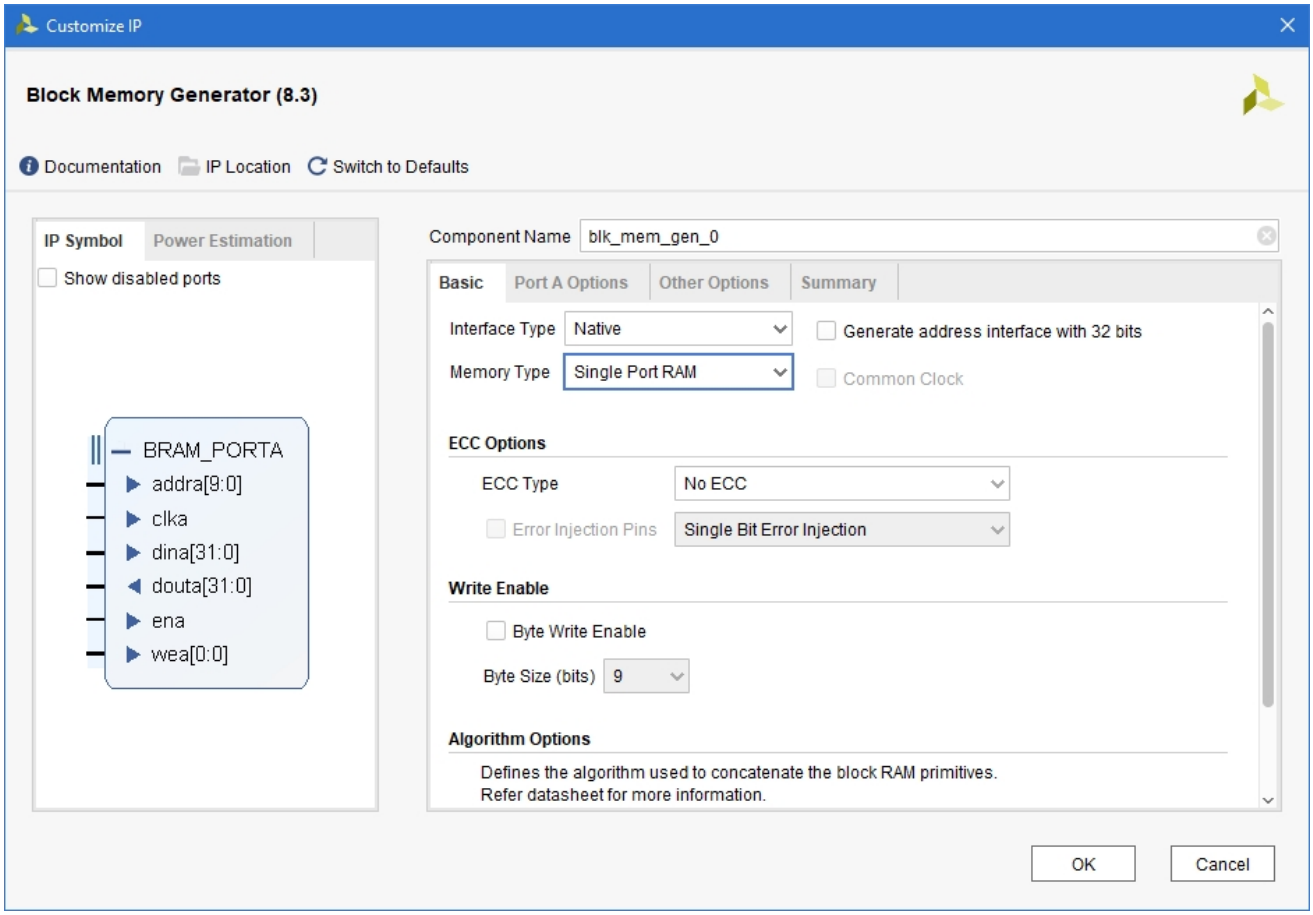

Figura 5.2: Quadre de diàleg per definir un mòdul d'accés a la *Block RAM*

1. Al camp *Component Name*, especifiquem un nom fàcil i amigable per anomenar al nou mòdul que es generarà.
2. A la pestanya *Basic*:
   (a) Al camp *Interface Type* cal especificar *Native*. D'aquesta manera ens crea una interfície de senyals estàndard que ens permetrà fer un disseny més llegible, fàcil de mantenir i portable en cas de voler executar el mòdul de l'arbre en un xip d'un altre fabricant.
   (b) Al camp *Memory Type* cal especificar *Single Port RAM*. D'aquesta manera indiquem que el contingut de la memòria és modificable i ens genera una interfície on indicarem les operacions de lectura i/o escriptura de manera seqüencial[2].

[2]Tenint present que es programa aquest mòdul "des de zero" i la metodologia de programació pròpia d'aquest sector, és altament recomanable que la primera versió sigui el més senzill possible i, en futures revisions, es pot millorar el comportament dels accessos a memòria utilitzant l'opció *True Dual Port RAM*.

3. A la pestanya *Port A Options:*

   (a) Al camp *Write Width*, cal especificar el nombre de bits que ocuparà cada posició de memòria. En el nostre cas, és el nombre de bits que ocupa un node de l'arbre. Aquest valor ha d'estar comprès entre 1 bit i 4608 bits.

      i. Cal assegurar que el valor del camp *Read Width* que aparegui sigui el mateix que l'especificat en el camp *Write Width*.

   (b) Al camp *Write Depth*, cal especificar el nombre de posicions que contindrà la memòria. D'aquest nombre, l'assistent ja s'encarregarà de generar el nombre de bits necessaris per indicar una adreça vàlida. Aquest nombre ha d'estar comprès entre 2 i 1048576.

      i. Cal assegurar igualment que el valor del camp *Read Depth* que aparegui a continuació sigui el mateix que l'especificat en el camp *Write Depth*.

   (c) Cal desactivar la casella *Primitives Output Register*. D'aquesta manera, escurcem la latència d'una operació de memòria a 1 cicle de latència[3].

4. A la pestanya *Other Options:*

   (a) Cal activar la casella *Load Init File* de la secció *Memory Initialization* per tal de poder indicar quin arxiu ha conté l'especificació de l'estat inicial de la memòria.

   (b) Cal prémer el botó *Browser*, per tal que obre un diàleg de sel·lecció d'arxiu per a indicar quin arxiu exactament ha de tenir present per omplir la memòria[4].

5.2.3 Ús del mòdul control·lador de la BRAM

El primer pas per tal d'emprar el mòdul que ha estat creat amb l'entorn Vivado és referenciar-lo i instanciar-lo en el mòdul on es vulgui instal·lar seguint el codi d'exemple de l'algorisme 5.1 a la pàgina següent. En segons lloc, si hi ha algun sub-mòdul que ha de fer ús d'aquest control·lador de la BRAM, s'ha d'ampliar la seva pròpia interfície d'acord amb l'exemple de l'algorisme 5.2 a la pàgina 64 per tal de poder comunicar amb el control·lador de la BRAM, ja sigui per realitzar una operació de lectura com l'exemple de l'algorisme 5.3 a la pàgina 64 o bé, per a realitzar una operació d'escriptura com en l'exemple de l'algorisme 5.4 a la pàgina 64.

5.3 Estructura d'un arxiu COE per a inicialitzar la memòria

Per tal de poder inicialitzar una memòria és necessari la creació d'un arxiu on s'indiqui els valors de la mateixa. L'eina Vivado de Xilinx utilitza arxius en format COE per a aquesta finalitat. Un arxiu coe és tot aquell arxiu de text pla amb extensió .coe i que conté els següents camps per inicialitzar una memòria:

1. *MEMORY_INITIALIZATION_RADIX*: aquest atribut indica la base sobre la qual es troben escrits els valors del contingut de la memòria ja sigui en binari (2), octal (8) o hexadecimal (16). Es finalitza la línia amb punt i coma.

2. *MEMORY_INITIALIZATION_VECTOR*: a aquest atribut s'assigna, separats per comes, la seqüència ordenada per posicions els valors que conformen el contingut de la memòria.

[3] Com abans, en futures revisions es pot marcar de nou aquesta casella si es creu necessari implementar un pipeline en les operacions de memòria per poder indicar en un port una operació de memòria diferent a cada cicle.

[4] En el següent capítol s'explica quina estructura té aquest arxiu.

Algorisme 5.1 Definició i instanciació del controlador de la BRAM

```
-- Aquesta deficnició del component s'introdueix en l'arquitectura del disseny on
    es defineixen les variables usades pels processos
component ram
    port ( clock : in std_logic;
        ena : in std_logic;
        wea : in std_logic_vector(0 downto 0);
        addr : in std_logic_vector(7 downto 0);
        din : in std_logic_vector(56 downto 0);
        dout: out std_logic_vector(56 downto 0)
    );
end component;
-- I aquesta instanciació va introduïda en la part on s'especifiquen els
   processos
bram: ram port map( clock => rellotge,
                    ena => permet,
                    wea => escriu,
                    addr => adreca,
                    din => dada_a_bram,
                    dout => dada_de_bram);
```

Un fitxer COE per a inicialitzar una memòria pot anomenar-se exemple.coe i contenir-hi la descripció mostrada en l'algorisme 5.5 a la pàgina 65:

Algorisme 5.2 Ampliació de la interfície d'un mòdul per afegir les senyals de control de la BRAM

```
entity redblack is
port ( rellotge : in std_logic;
        ordre : in std_logic;
        operacio : in std_logic;
        numero : in std_logic_vector (31 downto 0);
        fi : out std_logic;
        resultat : out std_logic_vector(7 downto 0);
        trobat : out std_logic;
        -- Senyals per control·lar la BRAM
        a_permet : out std_logic;
        a_escriu : out std_logic_vector(0 downto 0);
        a_adreca : out std_logic_vector(7 downto 0);
        -- El nombre de bits d'un node és 1 + 3*8 + 32 = 57
        a_llegir : in std_logic_vector(56 downto 0);
        a_escriure : out std_logic_vector(56 downto 0)
);
end redblack;
```

Algorisme 5.3 Operació de lectura a la BRAM

```
a_permet <= '1'; -- Indiquem que volem realitzar una operació
a_escriu <= "0"; -- Aquesta serà de lectura
a_direccio <= primer_element_ocupat; -- Indiquem l'adreça de l'element que ins
    interessa
-- ... Dos cicles de rellotge després
node_arrel <= (c => a_llegir(56), pare => a_llegir(55 downto 48), numero => (47
    downto 16), esquerra => a_llegir(16 downto 8), dreta => a_llegir(7 downto 0));
```

Algorisme 5.4 Operació d'escriptura a la BRAM

```
a_permet <= '1'; -- Indiquem que volem fer una operació
a_escriu <= "1"; -- Aquesta serà d'escriptura
a_direccio <= primer_element_ocupat; -- Adreça sobre la qual és fara l'escriptura
a_escriure <= negre & node_arrel.pare & node_arrel.numero & node_arrel.esquerra &
    primer_element_lliure; -- El node a escriure expressat com un vector de bits
```

Algorisme 5.5 Exemple de contingut d'un arxiu COE

```
memory_initialization_radix=16; Valors hexadecimals
memory_initialization_vector=
ffff,
0000,
0001,
a0bc,
1528,
0043,
fcab,
3689; Podem indicar els elements en diverses línies
```

Capítol 6

Resultats

6.1 Joc de proves

Es genera un fitxer de càrrega de treball de 1000 operacions, 250 insercions i 750 insercions i esborrats de nombres compresos entre l'1 i el 300 per forçar situacions d'inserció i esborrats d'elements existents a l'arbre. Per generar aquestes dades aleatòries s'utilitza el servei web Random.org[1].

6.2 Resultats de síntesis

Tot i que originalment es plantejava portar les implementacions descrites en el capítol 4 a la pàgina 33 a la FPGA Zynq 7020, es constata que no tots els dissenys hi tenen cabuda en aquell dispositiu. Llavors es decideix que els dissenys es portaran a la FPGA Zynq 7045, que és un dispositiu que pot encabir-hi dissenys més grans. Pero en la versió Webpack de Vivado no es troba disponible aquest dispositiu, però s'utilitzarà com a dispositiu target la FPGA Kirtex UltraScale 035, que té unes dimensions lleugerament inferiors al Zynq 7045.

Un cop s'ha triat el dispositiu target per a efectuar la síntesis, es realitza satisfactòriament la síntesis per als 6 dissenys donant com a números d'utilització de recursos els números que es mostren a la taula 6.1. S'hi pot veure com, a mesura que es complica l'estructura de dades a implementar, el nombre de recursos creix exponencialment. També es pot veure que la utilització de recursos decau espectacularment quan es porta les dades a la BRAM, fet que ens porta a la conclusió que el principal problema per encabir un dissenys en una FPGA determinada és el volum de dades que es pretén gestionar.

	Usa BRAM?			
	No		Si	
Estructura	#Luts	%Luts	#Luts	%Luts
Llista	9.216	4,54%	131	0,06%
Arbre binari	21.751	10,71%	413	0,20%
Red-Black	186.438	91,78%	1824	0,90%

Taula 6.1: Taula resum del procès de síntesis dels 6 dissenys el·laborats en aquest projecte

[1] https://www.random.org/

6.3 Resultats de la simulació

Es porta a terme l'execució del joc de proves en els 6 dissenys (3 estructures de dades, una versió no emprant BRAM i una altra emprant-la) en el simulador. Les primeres dades interessants són la millora de temps respecte a la llista: l'arbre binari és un 89,74% més ràpid en la versió sense BRAM o un 90,40% més ràpid en la versió emprant BRAM i l'arbre Red-Black és un 90,57% més ràpid en la versió sense BRAM o un 88,58% si utilitzem la BRAM tal i com es pot veure en la taula 6.2. Aquestes dades es poden justificar en el fet que hem passat d'una estructura amb operacions amb cost $O(n)$ amb operacions amb cost $O(\lg\ n)$ on *n* és el nombre d'elements en l'arbre o llista. L'altre fet destacat és que l'escriptura d'elements a la BRAM en sèrie i els cicles d'espera després de realitzar rotacions, provoquen que l'arbre Red-Black no millori en temps respecte l'arbre binari convencional.

Estructura	Sense BRAM	Amb BRAM
Llista	111987	333237
Arbre binari	11495	31982
Red-Black	10563	38065

Taula 6.2: Temps d'execució del joc de proves en cicles

Si desglossem les operacions en insercions i esborrats, podem veure en la taula 6.3, que les insercions es comporta millor l'arbre *Red-Black* i es torna a posar de manifest que les rotacions i l'ús de la BRAM por provocar que el temps d'una operació es dispari considerablement.

Usa BRAM?	Estructura	$mínim$	$mediana$	$mitjana$	$màxim$
No	Llista	3	119	107.65	178
	Arbre binari	4	11	11.73	21
	Red-Black	3	10	10.24	22
Si	Llista	5	354.5	320.65	533
	Arbre binari	8	32	31.68	62
	Red-Black	5	29	33.91	305

Taula 6.3: Nombre de cicles emprats en una inserció

Per altra banda, si ens fixem en les operacions d'esborrat de nodes en la taula 6.4, veiem com en la versió sense utilitzar BRAM l'arbre *Red-Black* els esborrats funcionen lleugerament millor respecte l'arbre binari de cerca convencional, però en la versió emprant BRAM en general els esborrats funcionen lleugerament pitjor però pot empitjorar molt el nombre de cicles degut a les rotacions i als escalats.

Usa BRAM?	Estructura	$mínim$	$mediana$	$mitjana$	$màxim$
No	Llista	4	148	119.62	178
	Arbre binari	5	12	11.78	20
	Red-Black	7	11	11.13	18
Si	Llista	8	441	355.42	530
	Arbre binari	11	32	32.51	60
	Red-Black	14	33	45.38	214

Taula 6.4: Nombre de cicles emprats en un esborrat

Capítol 7

Cost del projecte

7.1 Duració del projecte

Aquest projecte s'ha dut a terme entre els dies 9 de gener fins el dia 2 de juny del present any, amb la següents etapes i durada en hores:

Etapa	Durada en hores
Implementació Llista enllaçada	126 hores
Implementació Llista amb BRAM	42 hores
Implementació Arbre binari	168 hores
Implementació Arbre binari amb BRAM	42 hores
Implementació Red-Black	336 hores
Implementació Arbre Red-Black	126 hores
Total	840 hores

Taula 7.1: Repartiment del temps del projecte en etapes i la seva durada en hores

7.2 Cost econòmic

Es realitza un estudi de mercat per veure quin és el salari típic d'un programador en HDL. Tot i que en la majoria d'ofertes no apareix el sou de les ofertes, en la resta es pot veure que el sou mig brut anual d'un desenvolupador en HDL a Espanya és de 30.000€ aproximadament. Del qual es dedueix que els sou brut per hora treballada és de 14,423€/h treballada.

En funció de les hores emprades en la realització d'aquest projecte mostrades en la taula 7.1 es pot valorar, com es mostra en la taula 7.2 a la pàgina següent, aquest projecte per una valor de **14.659,54€.**

Concepte	Quantitat	Preu	Import
Hores treballades	840h	14,423€/h	12.115,32€
Impost sobre el valor afegit		21%	2.544,22€
Total			14.659,54€

Taula 7.2: Taula del cost total del projecte

Capítol 8

Conclusions

En l'el·laboració d'aquest projecte s'ha dut a terme satisfactòriament la implementació amb el llenguatge VHDL la llista enllaçada, l'arbre binari de cerca i l'arbre Red-Black. Addicionalment s'han creat les corresponents versions d'aquestes tres estructures de dades migrant la memòria de dades a les cel·les BRAM integrades en la FPGA, obtenint d'aquesta manera tres dissenys addicionals que empren menys recursos de la FPGA amb el cost de requerir més temps, en cicles de rellotge, per a realitzar una operació.

En el cas de la implementació de l'arbre Red-Black que usa la BRAM, es veuen unes oportunitats per tal de contra-restar l'estratègia conservadora emprada en la implementació per tal de reduir el nombre de cicles necessaris per a efectuar una operació. En un primer lloc, es pot dur a terme el sol·lapament dels cicles d'espera per BRAM, en operacions d'escriptura principalment, per la preparació d'altres operacions de memòria. Aquest canvi no implica cap requeriment especial sobre la FPGA ni cap ampliació de recursos per part de la mateixa. Posteriorment, aprofitant en aquest cas que s'utilitza una FPGA amb més recursos, es pot substituir el mòdul de control de la BRAM per una altre amb dos ports. Aquest fet implicaria la possibilitat d'efectuar en un mateix cicle de rellotge dos operacions a memòria, reduint d'aquesta manera els cicles consecutius que efectuen en cada un una operació de memòria sense cap mena de dependència. Aquest canvi seria molt profitós en les re-estructuracions de l'arbre, és a dir, en les rotacions i els re-pintat dels nodes.

Bibliografia

[1] X. Tan, J. Bosch, D. Jiménez-González, C. Álvarez Martínez, E. Ayguadé, and M. Valero, "Performance analysis of a hardware accelerator of dependence management for task-based dataflow programming models," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016, pp. 225–234.

[2] F. Yazdanpanah, C. Álvarez, D. Jiménez-González, R. M. Badia, and M. Valero, "Picos: A hardware runtime architecture support for ompss," *Future Generation Computer Systems*, vol. 53, pp. 130 – 139, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X14002702

[3] M. G. Arnold, *Verilog digital computer design: Algorithms into hardware*. Prentice-Hall, Inc., 1998.

[4] K. Nagasu, K. Sano, F. Kono, N. Nakasato, A. Vazhenin, and S. Sedukhin, "Parallelism for high-performance tsunami simulation with fpga: Spatial or temporal?" in *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*. IEEE, 2016, pp. 30–30.

[5] E. Ayguade, R. M. Badia, P. Bellens, J. Bueno, I. T. Teruel, and M. Valero, "Hybrid/heterogeneous programming with ompss and its software/hardware implications," *Programming Multicore and Many-core Computing Systems*, vol. 86, p. 101, 2017.

[6] N. Bell, S. Dalton, and L. N. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C123–C152, 2012.

[7] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas, "Kilocore: A fine-grained 1,000-processor array for task-parallel applications," *IEEE Micro*, vol. 37, no. 2, pp. 63–69, 2017.

[8] T. Dallou, D. C. S. Lucas, G. Araujo, L. Morais, E. F. Barbosa, M. Frank, R. Bagley, and R. Sayana, "Task parallel programming model+ hardware acceleration= performance advantage," in *Hot Chips 28 Symposium (HCS), 2016 IEEE*. IEEE, 2016, pp. 1–1.

[9] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: architectural support for fine-grained parallelism on chip multiprocessors," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 162–173.

[10] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Enhanced architectures, design methodologies and cad tools for dynamic reconfiguration of xilinx fpgas," in *Field Programmable Logic and Applications, 2006. FPL'06. International Conference on*. IEEE, 2006, pp. 1–6.

[11] I. Kuon, R. Tessier, and J. Rose, "Fpga architecture: Survey and challenges," *Foundations and Trends in Electronic Design Automation*, vol. 2, no. 2, pp. 135–253, 2008.

[12] S. Brown and J. Rose, “Fpga and cpld architectures: A tutorial,” *IEEE design & test of computers*, vol. 13, no. 2, pp. 42–57, 1996.

[13] “Ieee standard for verilog hardware description language,” *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pp. 1–560, 2006.

[14] “Ieee standard vhdl language reference manual,” *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pp. 1–626, Jan 2009.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2009. [Online]. Available: http://www.ebook.de/de/product/8474892/thomas_h_cormen_charles_e_leiserson_ronald_l_rivest_clifford_stein_introduction_to_algorithms.html

[16] P. Ashenden, *The Designer’s Guide to VHDL*. Elsevier LTD, Oxford, 2006. [Online]. Available: http://www.ebook.de/de/product/7336161/peter_ashenden_the_designer_s_guide_to_vhdl.html

[17] “Ieee standard multivalue logic system for vhdl model interoperability (std_logic_1164),” *IEEE Std 1164-1993*, pp. 1–24, May 1993.

[18] L. H. Crockett, R. A. Elliot, and M. A. Enderwitz, *The Zynq Book*. Strathclyde Academic Media, 2014. [Online]. Available: http://www.ebook.de/de/product/23268239/louise_h_crockett_ross_a_elliot_martin_a_enderwitz_the_zynq_book.html

[19] ——, *The Zynq Book Tutorials for Zybo and ZedBoard*. Strathclyde Academic Media, 2015. [Online]. Available: http://www.ebook.de/de/product/25469806/louise_h_crockett_ross_a_elliot_martin_a_enderwitz_the_zynq_book_tutorials_for_zybo_and_zedboard.html

[20] A. G. Olivo and J. P. Martínez, *Diseño de procesadores con VHDL*. Universidad de Alicante. Servicio de Publicaciones, 2007. [Online]. Available: http://www.ebook.de/de/product/12575575/angel_grediaga_olivo_jose_perez_martinez_diseno_de_procesadores_con_vhdl.html

[21] D. Thomas and P. Moorby, *The Verilog® Hardware Description Language*. Springer Science & Business Media, 2008.